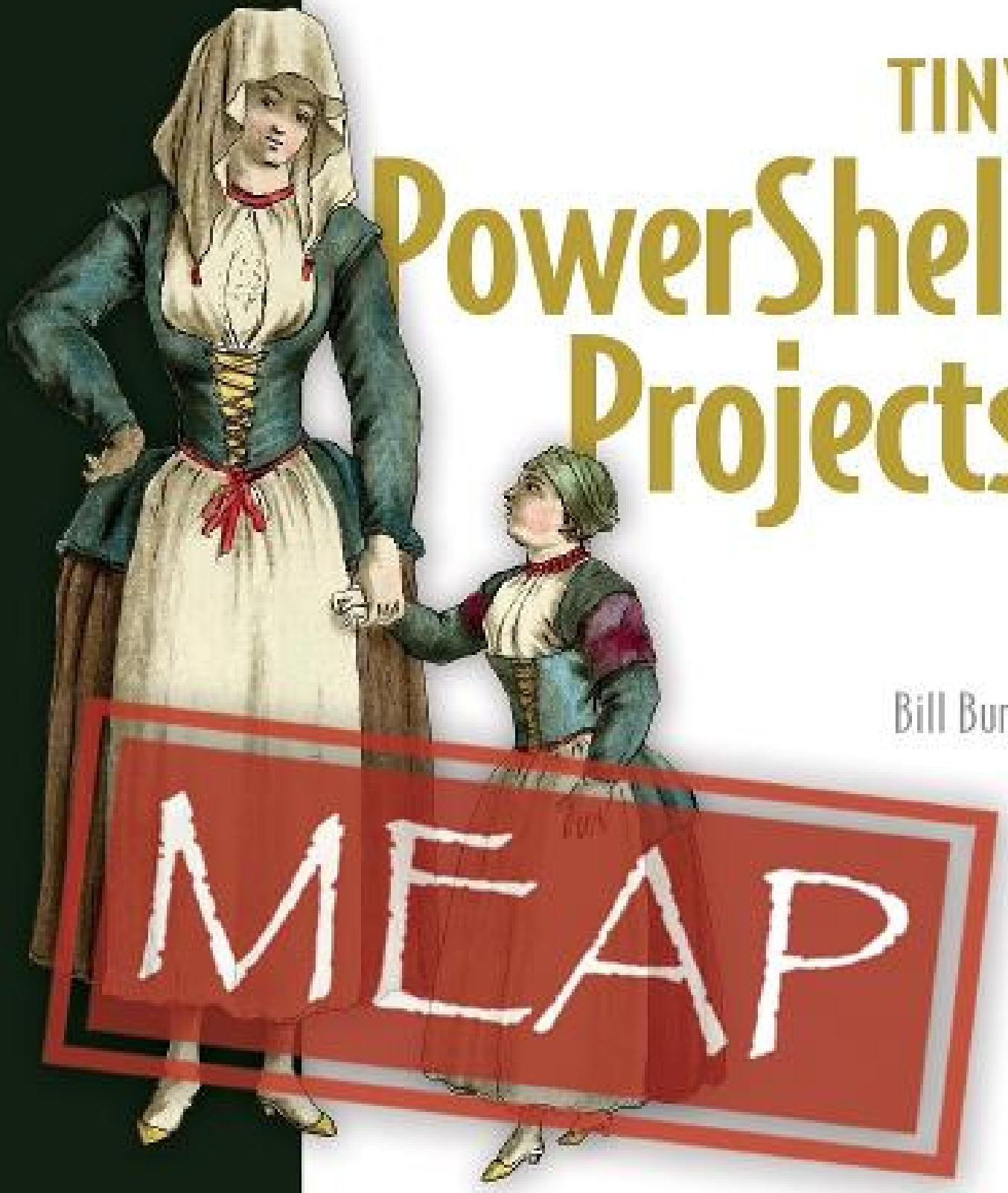
TINY PowerShell Projects



Bill Burns

MANNING

Burns Tiny PowerShell Projects MEAP V06

MEAP Edition

Manning Early Access Program

Tiny PowerShell Projects

Version 6

Copyright 2023 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes.

These will be cleaned up during production of the book by copyeditors and proofreaders.

https://livebook.manning.com/#!/book/tiny-powershell-projects/discussion

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for purchasing the MEAP for *Tiny PowerShell Projects*.

This book has been written for those IT Professionals with little to no coding experience. But even those with a firm understanding of the language can hopefully find some useful scripts that will drastically improve their ability to automate and optimize their environment.

PowerShell, perhaps more than any other language, is the perfect language for the IT Professional to learn. It has power, flexibility and an ease of entry that is not found in any other tool.

There are a great number of IT Professionals who are, at best, underutilizing this powerful tool. This, I believe, comes from the simple fact that many IT Professionals are not coders and yet nearly every book on the subject is written to teach the language like you would to a software engineer.

Rest assured; this is not the nature of *Tiny PowerShell Projects*. I have never considered myself a software engineer. I first experienced PowerShell, nearly ten-years ago, while working as a remote support technician for a large company. My first PowerShell course was taught like it was any other programming language. We covered variables, if statements and loops, I spent five days in a live training program, but frankly I found myself in debugging hell because I'd do something like forget to end my closing brace. I quickly found that I was being buried in the newness of it all. Relieved when I could type a string of commands into the interpreter and not see a screen full of red staring back at me. I wasn't getting anything out of it that was useful. I could construct some code and make it run, for the most part. But it was a long and arduous process, and I still had no idea how, when, or why to use PowerShell in my daily life.

My goal for *Tiny PowerShell Projects* is to flip this paradigm. I will start with the why, then show you the when and only then, will I detail the how.

Tiny PowerShell Projects is designed to produce useful scripts and demonstrate use cases that you can take out of the book and put directly into practice in your environment. Then, I will break down the script section by section covering the language and logic that make it work. It is quite simply the book I wish I had had when learning PowerShell.

Feedback on this project has already been incredible and inspiring. I hope that you'll be leaving comments in the liveBook Discussion forum. Your question, comment or suggestion may help future readers as they too discover this powerful automation and optimization tool at their fingertips.

Thanks again for your interest and for purchasing the MEAP!

—Bill Burns

In this book

1 Introduction to PowerShell 7.x

This chapter covers:

- What is PowerShell?
- Why is it important?
- How do I use this book?
- Who should know PowerShell?
- What's the difference between PowerShell 5.x and 7.x?
- An example of why learning PowerShell makes your job easier

If you have picked up this book, congratulations! You have just found a resource that will not only make your life easier; it will make you more efficient and more marketable. PowerShell is a remarkable tool for I.T Professionals. In the industry, there is a line being drawn, those Admins who can automate tasks and those that do them the hard way.

Look at all of the systems that have come to dominate the industry. WSUS, SCCM, Jenkins and Kubernetes just to name a few. If you look at what these tools do, the one thing they have in common is taking things we would have previously done by hand, and instead automating them at scale. Automation is quite simply the most important resource an I.T. Professional can learn today to make them relevant in the future; and PowerShell is one of the best ways to automate your domain.

1.1 What is PowerShell?

PowerShell is a task automation and configuration tool built into all Microsoft Operating Systems since Windows 7. PowerShell was such a useful tool that versions of PowerShell for XP and Windows Server 2003 were released as add-ons to the operating systems in 2009. But PowerShell is more than this. Unlike earlier Microsoft task automation and configuration tools PowerShell is a fully defined language utilizing the powerful .Net framework Object Orientated Programming (OOP). With its pre-defined cmdlets native to Microsoft systems, servers and domains an

admin doesn't need to have any previous programming experience to make it useful right away; and as your programming experience grows so too does the flexibility and power of the language. A cmdlet (pronounced command-let) is a lightweight command written into the PowerShell language that can be used to perform specific functions. For example, you don't need to know how to write a program to add a computer to a domain. You can simply type New-ADComputer (an Active Directory cmdlet) and PowerShell understands what you are wanting it to do: adding the PC to your domain. But this flexibility is not just limited to Active Directory many Microsoft products and even non-Microsoft products implement cmdlets to their specific technology to allow for automation of administration and configuration

Competency in this language will change the way System Administration is done. Yet, it offers enough flexibility and power that those who deep dive into this language can gain nearly limitless control over their entire domain. SharePoint, Exchange and Office365 just to name a few, all support and often utilize PowerShell for their own backend Graphical User Interfaces (GUI)s.

1.2 Why is it important?

Windows PowerShell, more than any other language, can have an immediate impact on your ability to truly master your domain. As IT professionals we all have tasks that need doing repeatedly. It could be creating users, making reports, building mailboxes or any number of things. Let's examine just a single one of these common tasks: checking to see the status of a service on all of your Windows systems.

The non-automated way of doing it requires:

- Acquiring a list of hosts
- Manually connecting (Remote Desktop, PSExec, WMI) to each host
- Querying the Service
- Updating the list

On a handful of systems this is an inconvenience. On several dozen this is an all-day time suck. On several hundred this is a massive project! This task, and many more, can be done with PowerShell on several thousand hosts with just a few minutes of coding and completed on the whole domain by lunch. This book will provide the code needed to do exactly this type of task!

If you're still not sold on why a System Admin, or anyone wanting to maximize their ability to get work done on a Windows System should be utilizing PowerShell, imagine if at the end of that same task your boss came to you and said that you now needed to enable the service you checked for on all computers that didn't have it running and check them all for a second service as well. You, the old school System Admin are back to square one. Your PowerShell proficient co-worker can get the same task done in hours you would speed weeks doing. Wonder who's getting the bigger raise? It's time to give yourself the unfair advantage you always dreamed of. This book is the first part of that journey.

1.3 How do I use this book?

This book is written as a series of Tiny PowerShell projects. Each of these projects will do something useful in your Windows environment; something you can implement today and see optimization in your environment even if you don't initially understand how the project works or why.

Each project is then dissected, explaining the code that makes the script work. Detailing critical programming concepts like variables, functions, branching, loops, structured error handling, arrays, and methods. You don't need to have any kind of programming background to see the value of implementing the projects in your environment and using them as written.

If, however, you have some experience with PowerShell the projects in this book will be immediately useful and you can enhance your skills with practical hands-on projects. You'll see, through the application of these projects, many more potential opportunities to automate and customize your own domain making your administration task lists less about the repetitive tasks and more about the true value add you bring to your organization.

If you have no desire to learn anything about PowerShell and how it works, you'll still get value add in the form of the seventeen specific and useful PowerShell scripts you can implement today to improve your productivity and proficiency in the language. It's my goal not to waste anyone's time with scripts that do not serve a tangible improvement to your environment. You won't find a "hello world" script in the book. That being said, the beginning scripts will ease new PowerShell users into the language and syntax at a pace designed not to overwhelm them. More advanced users should read the script at the beginning of the chapter. If you understand exactly what and how the script functions, feel free to skip the dissection of the script in the rest of the chapter. Many times, future scripts will build upon the basics we learned in earlier chapters to achieve more complicated solutions.

But should you see the power and flexibility native to the language this book will serve as a valuable jumping off point illustrating not only what can be done to improve your life as a System Admin, Exchange Admin, Security Personnel, or other IT professional, whether you are working on premises or in the cloud. It will also show you how to recognize the most advantageous times to implement a scripted solution.

1.4 Who Should know Windows PowerShell?

System Administrators, specifically Windows System Administrators, a lot to gain from a deep understanding of Windows PowerShell. This is because they have the most flexibility to utilize the cmdlets built into Windows Systems, Servers, and Applications. However, since this book is written for PowerShell 7.1, even non-Windows admins can see a great deal of value implementing these scripts. PowerShell 7.1 is natively compatible with Windows 8.1, Windows 10, Windows Server Versions 2012-2019, MacOS 10.13+, Red Hat Enterprise Linux and CentOS 7, Fedora 30+, Debian 9, Ubuntu LTS 16.04+, and Alpine Linux 3.8+.

Exchange Admins will see a tremendous advantage in mastering Windows PowerShell. This is because Microsoft Exchange functions on a PowerShell backend. Anything that can be done in the GUI is translated into PowerShell commands that execute on the server. Knowing the cmdlets and Command

Line Interface (CLI) will make you easily an order of magnitude more efficient than those Admins who are forced to rely on the Web Interface GUI to do the same thing.

Information Security Professionals would also benefit from the ability to traverse the network, parse logs, create reports and deploy critical and zero-day patches to your domain on the fly.

In general, anyone, even a user, who has a repetitive task could benefit from the automation and configuration capabilities of Windows PowerShell.

Ironically it is the group of Professionals that have the most to gain from understanding Windows PowerShell that are the most resistant to using it. This, I believe, comes from the simple fact that System Administrators are not traditionally coders. If you have ever been put off learning PowerShell or have felt intimidated, or don't know where to start, this book is for you! You have come to the right place. Each tiny project is useful and your hands-on experience will help you overcome your fear and dive into the amazing world of Scripting and Automation!

This is where Tiny PowerShell Projects takes a different approach. By utilizing useful, tangible, and functional code projects that the Administrator can implement into their environment and dissecting how they work even those without a comfort or immediate interest in the language can see an immediate benefit.

1.5 What's the difference between PowerShell 5.x and 7.x?

As of the time of writing this book the most current version of PowerShell is PowerShell 7.2. However, PowerShell 7.x does not come pre-installed on any version of Windows or Windows Server, to date. To understand this, you need a little background in the recent history of Microsoft PowerShell.

Prior to 2018 PowerShell was built on the .Net Framework system. This gave PowerShell a lot of added flexibility because if a cmdlet did not exist for your current need you could essentially program your own utilizing

languages like C# (Pronounced C Sharp) or VB.Net or anything that utilizes the .Net framework. This made the reach of PowerShell virtually unlimited on a Microsoft System. However, it also meant that PowerShell would be relegated to operate largely in Windows only environments. .Net Framework was only functional, natively, in Microsoft operating systems.

In 2018 Microsoft sought to change this. They developed .Net Core which was a cross platform version of the .Net Framework. This enabled functionality between Microsoft products, MacOS and Linux flavors. PowerShell Core 6.0 was developed to utilize the .Net Core. The current version of PowerShell, PowerShell 7x is a continuation of this cross-platform development.

However, to implement cross-functionality, PowerShell 7x has introduced some case sensitivity that might affect previously written scripts where this was earlier not important. One small example of this is case sensitivity. Both Linux and MacOS tend to be case sensitive. Windows, however, is largely case insensitive.

In addition to this simple example a large number of cmdlets normally native to PowerShell make no sense in a Linux/MacOS environment and thus have been omitted in PowerShell 7.x. These cmdlets deal largely with registry keys and processes but also have an impact on NTFS file permissions. Microsoft has made some significant strides in backwards compatibility between PowerShell 7.x and PowerShell 5.1, including ways to import a PowerShell 5.1 session into your PowerShell 7.x environment to run all of these normally native cmdlets within PowerShell 7.x.

All in all, although there may be some issues moving some of your existing scripts to a PowerShell 7.x environment, something this book will address in later chapters, PowerShell has given the typical I.T. Professional a very powerful toolset to automate and optimize their environment in many ways. Simply put, I.T. Professionals who do not learn this powerful skillset will soon no longer be able to keep up with those who do.

Choosing a Version of PowerShell

Consider staying PowerShell 5.1 if:

- You are a Windows-only shop with a small number of PCs and servers
- You are unable to upgrade to .Net Core
- You cannot run an external ISE like VSCode
- You have a lot of critical legacy PowerShell code

Consider PowerShell 7.x if:

- You are dealing with a mix of OSes
- You are utilizing websites such as StackOverflow
- You are utilizing VSCode
- You have a medium-size or larger enterprise
- You have a small amount of critical legacy PowerShell code or can spend the time troubleshooting issues legacy code might have operating in a 7.x environment.

1.6 How does this make a System Admin's life easier?

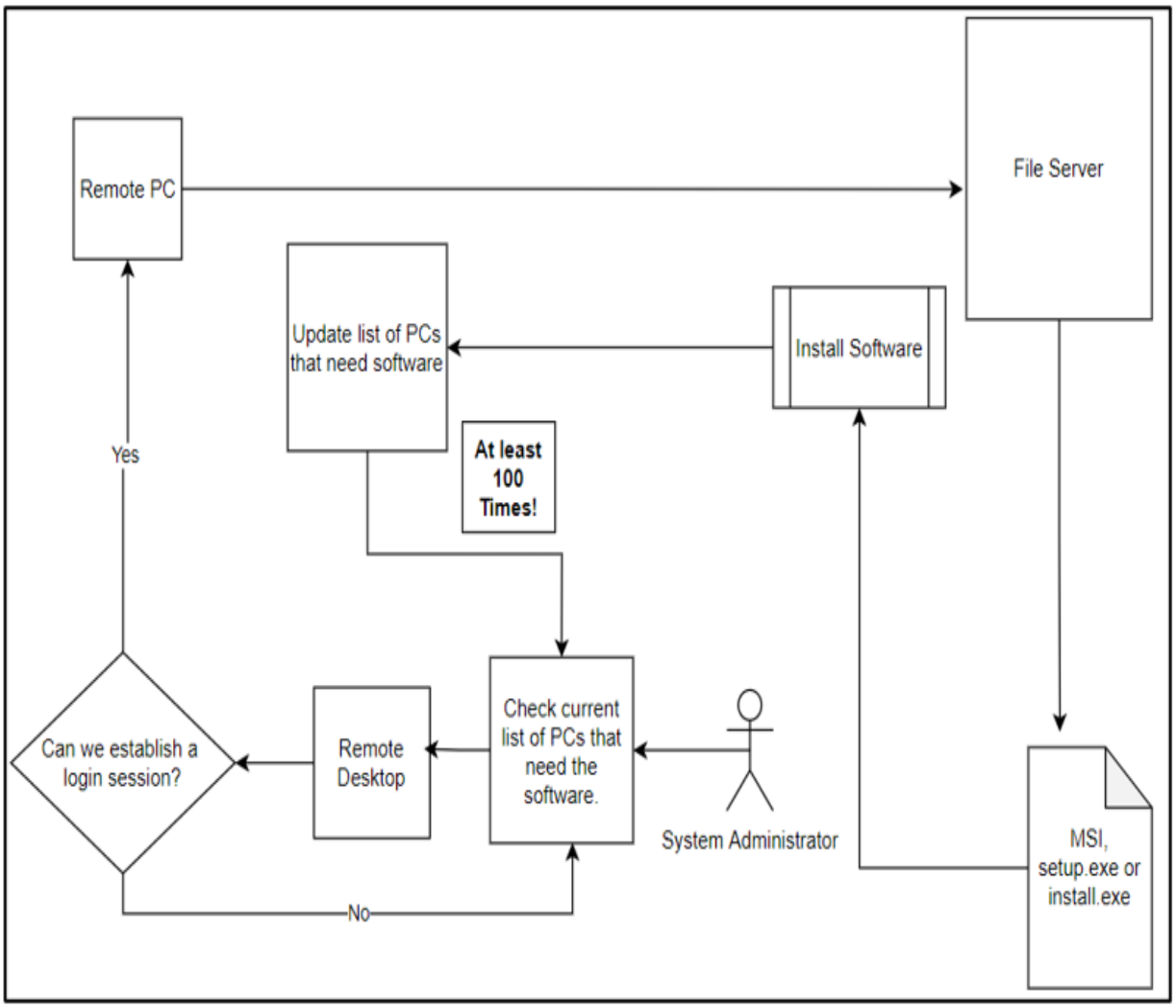
There are already a number of widely used tools that make the task of controlling your domain easier: SCCM, GPO, CHEF, Puppet or Ansible just to name a few. Yet even with all of these tools available to a I.T. Professional, there will undoubtedly be times where PowerShell is the right tool for the task at hand. Say for example there was a software package you needed to install on 100 PCs to address a zero-day vulnerability. Let's look at how you would do this the old-school way.

1.6.1 The traditional way includes:

- Creating a list of PCs that need the software. This in itself may take several minutes to several hours depending on how you are keeping track of your computer inventory.

- A System Admin then remotes into each computer on the list.
- With a Windows Operating System, it's likely that only a single active session can exist on the PC at any given time. If the computer is in use, it's likely that the System Admin will be unable to remote into the PC to install the software. This means that the PC is likely skipped and will have to be re-tried.
- If the System Admin successfully accesses the system, the user profile is created or updated (this may take several minutes).
- Once they are in the active session on the PC, they can now navigate to the file server and manually install the software. This can take several minutes or a very long time depending upon the network speed, disk speed, and size of the program.
- If successful, the System Admin updates their list manually and moves to the next.
- This process is repeated at least 100 times. Keep in mind that every PC a System Admin cannot access immediately will need to be retried. This can take several attempts for the System Admin and the downtime to the company for the install can be a significant impact.

Figure 1.1 Installing software on 100 Windows PCs the old way.



Now, let's examine how using PowerShell 7.x improves this situation for the System Admin, the users, and the company.

1.7 The PowerShell way includes:

- The System Admin runs the install script.

That's it.

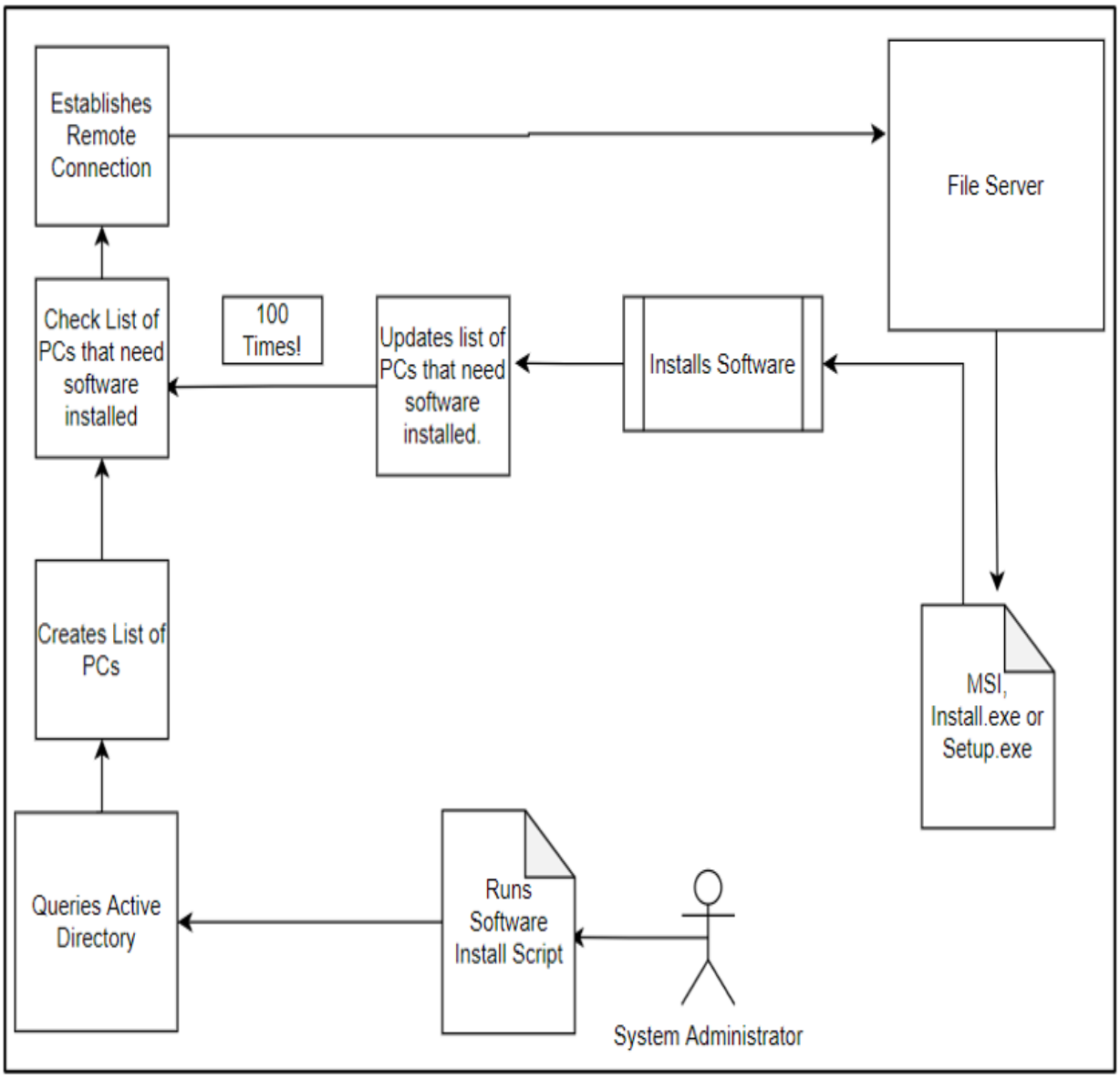
To understand, let's look at how PowerShell is accomplishing all of this:

- PowerShell has built in compatibility with Active Directory. You can use Active Directory and PowerShell to build your list of targets,

finding all Active computers in a given Active Directory Organizational Unit (OU), the whole domain or any combination of OUs. You can filter on any attribute the PC object has in Active Directory like Operating System, Member Of, Location, Managed by or many others.

- PowerShell can check to see if the system is responding to pings, targeting only those systems that are connected to your network (not turned off, or off the network, like a laptop).
- PowerShell can multithread, running multiple parallel processes, seamlessly remoting and installing your software on more than one PC at a time.
- PowerShell can install your software, accepting argument flags like -qn or -silent to install your software without requiring an active session. More on these flags when we cover using PowerShell to install software on remote computers in later chapters.
- PowerShell can keep track of successes and failures for the software installation attempt and create a report of PCs that still need the software installed.
- This report can then be utilized as the targets for any follow-up attempts to run this software on the outstanding PCs without having to target the whole domain or OU list again.

Figure 1.2 Installing software on 100 PCs with PowerShell



All of this functionality can be written in minutes and could be run and re-run on-demand. This book will give you working examples of all of this behavior that you, as a I.T. Professional, with little to no coding experience, can use to construct this exact type of script.

1.8 Summary

- PowerShell is a task automation and configuration tool built by Microsoft.

- PowerShell utilizes cmdlets (command-lets) to make it immediately useful to I.T. Professionals with little to no coding experience.
- Cmdlets are lightweight PowerShell commands that allow you to perform a wide range of tasks across your entire I.T. ecosystem.
- PowerShell will allow you to complete many common or repetitive tasks with ease, making you many times more efficient than those I.T. Professionals who have yet to embrace this skillset.
- This book will contain a series of Tiny PowerShell Projects that you can use on your systems today that will give you immediate and tangible value and teach you the language at the same time.
- There are significant differences between the legacy Windows PowerShell 5.1 and PowerShell 7.x. The current future of PowerShell is 7.x and will be the focus of this book.

2 Automating email address creation

This chapter covers:

- Using Variables
- Checking Variable contents with Write-Host
- Removing white space from strings
- Selecting parts of a string
- Modifying the capitalization of whole strings

Automating tasks as an I.T. Professional is the first step in freeing yourself from the menial so that you can focus on the value adds to your organization. Every company needs basic I.T. tasks done. But the companies we work for often have much larger and more pressing demands on our time.

System outages, software upgrades, hardware upgrades, security concerns or even standing up entire remote locations. These types of tasks are frequently on an I.T. Professional's plate. Far too often an administrator is pulled in every direction at once and your days can feel like you've simply jumped from one fire to the next.

While it may sound simple to build a user account in Active Directory Users and Groups: *"That will only take me five minutes."* It is frequently five minutes we don't have when seen in the grand scheme of things.

Automation is a way of taking the simple things off your list, letting the computer do the work, so you can focus on the things that allow you to be more productive, more valuable, and less likely to cause burn out.

Note

This book builds upon the assumption that the reader has little to no previous programming experience. Therefore, it is necessary to start slow and build upon the reader's skillset, building to more and more complex projects. While it is the aim of this book to provide value in every Tiny PowerShell project script, some of the early chapters will be less powerful for those I.T. Professionals with a firm grip on the PowerShell language. This script provides inherently more value to the admin than running a "Hello World" would. However, it is something that might seem trivial for a professional with a firm understanding of all the tasks within the script.

If you already have a firm grip on the foundations, please feel free to give the script a quick look and progress to more advanced chapters.

Nearly every company has email that they use for a way of communicating with their employees. Thus, an email address is almost always something that you will need to know. Frequently they are formulaic.

First letter of the first name "dot" Last name @ Domain name "dot" com. Still, building a list of email addresses for quick communications, reports or requests can be time consuming when given a list of users. We will explore much more efficient ways to pull this information in upcoming chapters but for now, let's look at how we might use PowerShell to help us with the task.

If we are given a list of names to get an email address from the list we would need to:

- Take the first initial of the first name
- Combine it with the last name
- Remove any blank spaces before or after the names
- Add an "@" symbol to the end
- Finally add the domain name

Even a skilled typist, which many System Administrators are not, would take several seconds to perform this function manually. The opportunity for typos is quite large when faced with a large list. How often do we insert a ".com" at the end of a domain name by wrote (even if our own organization is a .org or .edu)?

This brings us to our very first Tiny PowerShell Project! This script will:

- Allow us to assign a First and Last name to variables. You can just copy and paste this from your list.
- Allow us to assign a domain name (that we only have to type once, so the likelihood of confusing a .org with a .com goes nearly to zero)
- Cleans up any white space, so you don't need to worry about cleaning your list before you copy and paste. A leading or trailing space, or even several, will not make it into the email address output
- Finds the first initial of the first name
- Writes out the full email address. (First Initial+Last Name+@+domain name+.com)

2.1 Project Code: Automating email address creation

Let's take a look at our first script below:

```
$First_Name="John "
$Last_Name=" Doe"
$Domain="@ForTheITPro.com"
$Email_Address=     $First_Name     .Trim().SubString(0,1)+
$Last_Name     .Trim()+$Domain
Write-Host $Email_Address
```

The rest of the chapter will walk you through what each step of the script does and how it works.

2.1.1 How to run the script

The first thing we should discuss is how to execute these PowerShell scripts. With the legacy Windows PowerShell 5.1, Microsoft included an Integrated Scripting Environment (ISE) to run PowerShell. However, Microsoft already has arguably the leading Integrated Development Environment (IDE) on the scene today with VS Code.

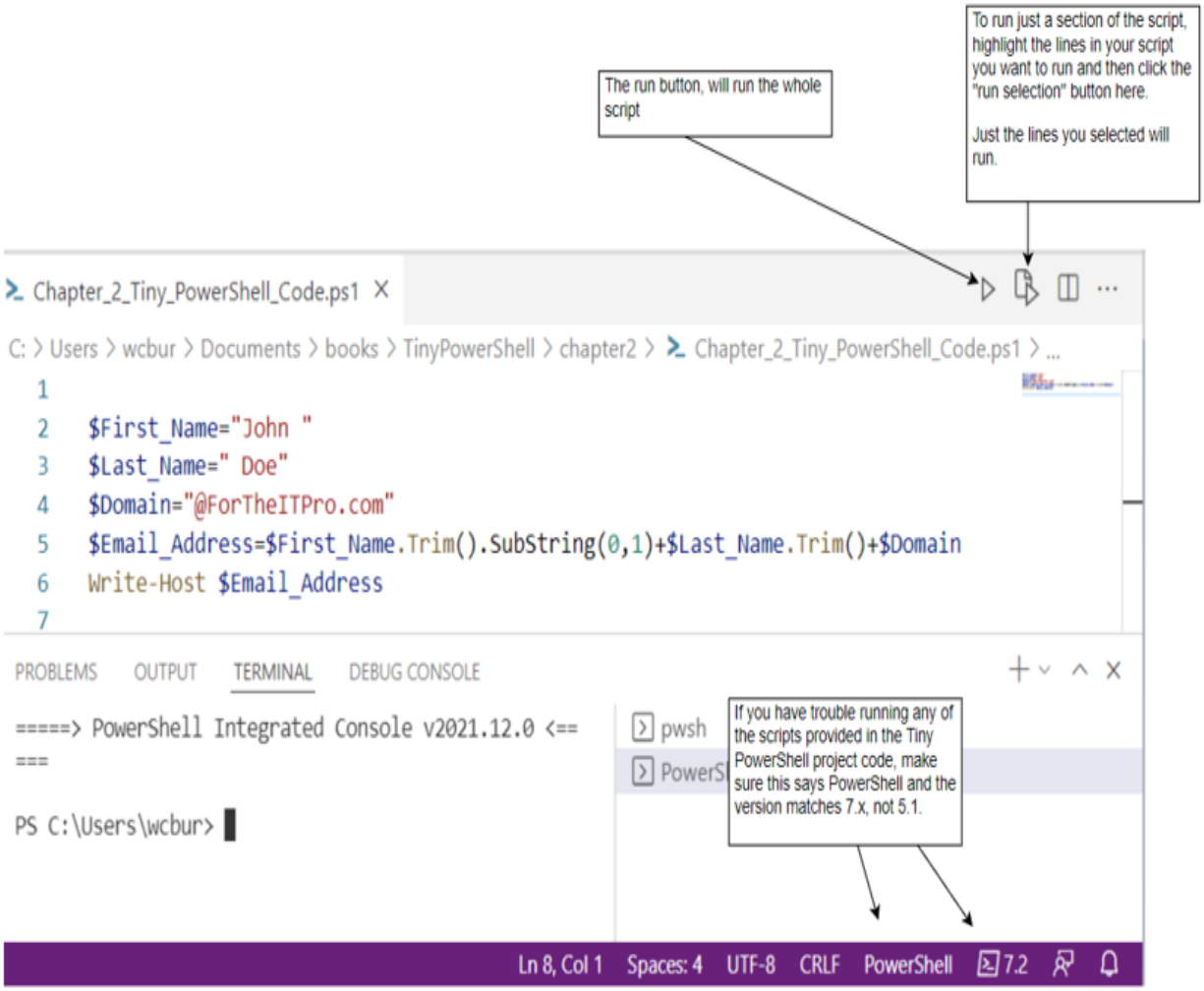
VS Code already had built in PowerShell support, so instead of supporting two scripting Environments Microsoft simply stopped support on

PowerShell's ISE with the legacy Windows PowerShell 5.1. It is strongly recommended that you download and Microsoft's VS Code IDE and use it for the scripts within this book.

There are step-by-step instructions in the README.md included with the scripts for the Tiny PowerShell project code.

Once you have VS Code and PowerShell 7.x installed on your system, simply open VS Code and you can select File → Open File → and navigate to the script you want to open. VS Code will understand that the .ps1 file is run with the PowerShell 7 software you installed.

Figure 2.1 How to run a script in VS Code.



2.1.2 What does this script do?

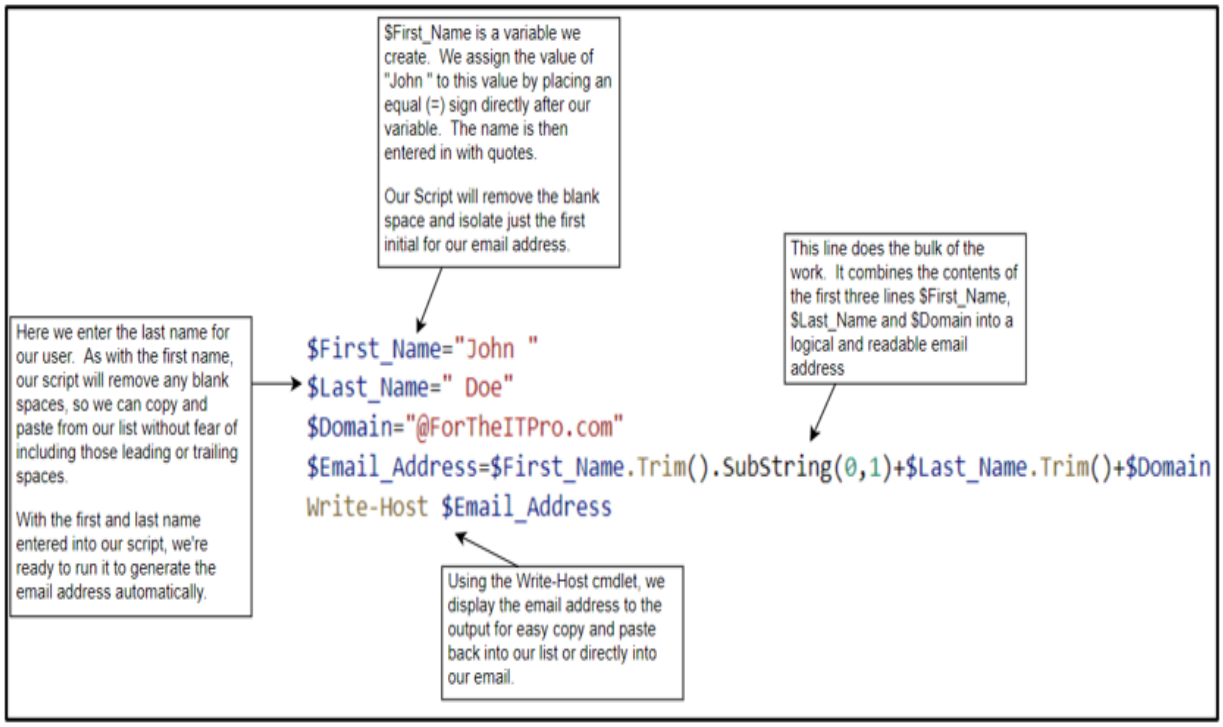
The Project Code can be used to generate email addresses. To accomplish this, the script declares (creates) the following variables (data placeholders discussed later in the chapter) and assigns them values: First_Name, Last_Name, Domain and Email_Address. To declare a variable in PowerShell you simply use the dollar sign ($) immediately followed by the name of your variable.

The equal sign (=) is an assignment operator in PowerShell; whatever is on the right-hand side of the equal sign is assigned as the value of the variable on the left-hand side. If you want the First_Name variable to have the value "John", you use the equal sign (=) to do it, which is what the first line does.

By modifying the `$First_Name` and `$Last_Name` variables and setting the $Domain to the proper email address type you can generate email addresses for your domain.

The script will automatically remove white space, trim the `$First_Name` to the first initial and concatenate *the* `$First_Name`, `$Last_Name` and `$Domain` into an email address format.

Figure 2.2 Project Code for Automating Email Address Creation.



2.2 What's a string?

When you are dealing with data it's intuitive that not all data is treated the same. You can't, for example, add "John" and "Doe". These are not numbers (Integers) so trying to add them doesn't make any sense. Instead, both "John" and "Doe" are words made up of a string of characters, or in programming terms a `string`. There are many different data types used in most programming languages, and PowerShell is no exception. However, unlike a great deal of other languages, PowerShell is an optionally strongly-typed language. This means that we, as the writer of the script, do not need to pre-define the types of objects we use in our variables. PowerShell, like any other programming language, still needs to know what type of data you are using. Including quotes, single or double (' ') or ("") around a string of letters is the best way to indicate to PowerShell that this value is intended to be a string.

Definition:

PowerShell Cmdlets: A Cmdlet (pronounced command-let) is a command loaded into PowerShell at runtime. These cmdlets are usually loaded by PowerShell itself or by importing MODULES (more on that later in this chapter). These cmdlets simply let us interact with PowerShell and the APIs to interact with our Windows environment, our PowerShell ISE, or Remote Computers.

2.3 Variables

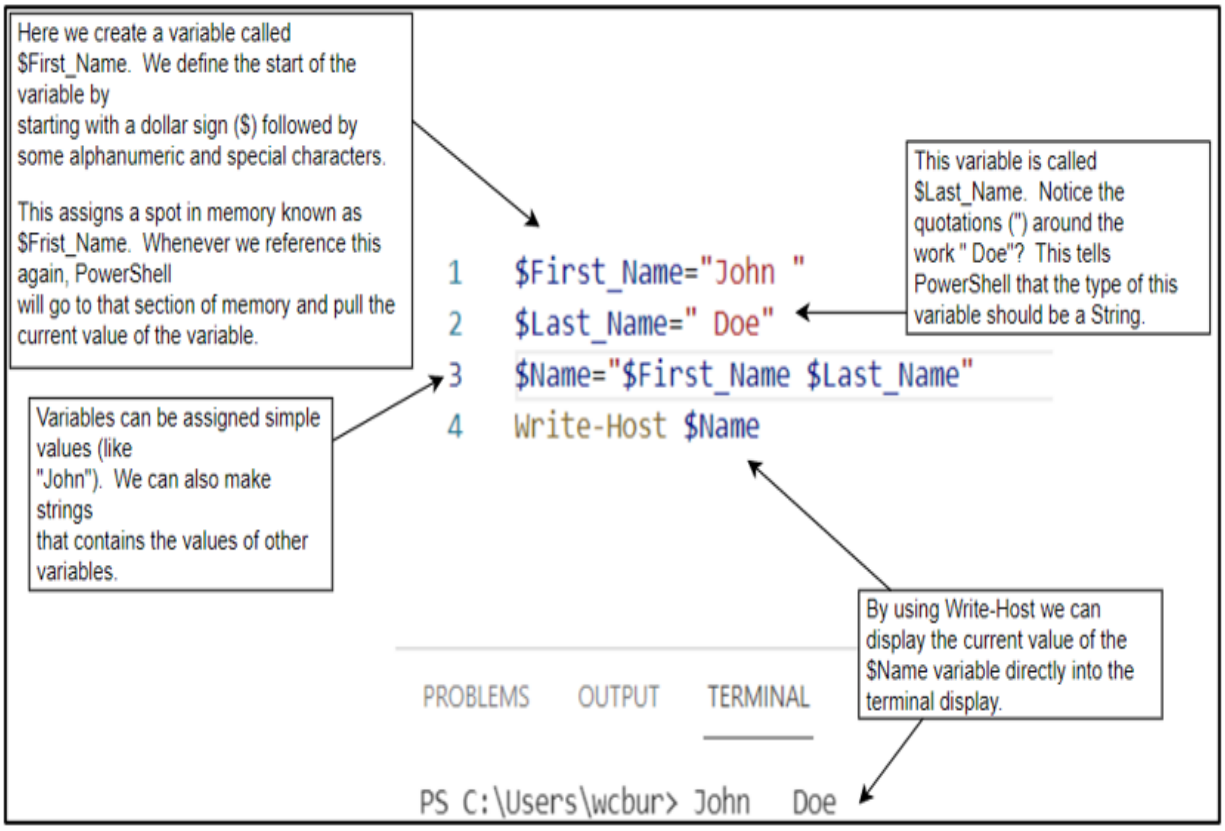
Variables are simply placeholders for data. Think of a variable like a shoebox. You can put all kinds of stuff in a shoebox, or nothing at all. You could put virtually anything you like in your shoebox and get it later; and unless you replace the contents with something else your shoebox will have the same thing in it you put there when you go to get it later. In PowerShell a variable is defined with a $ followed by some alphanumeric or special characters. Variables names are not case sensitive, they can start with special characters, letters or numbers. Variables can be named almost anything you like, but they should be called something that generally explains the type of information they will be holding. If you have a variable to hold a first name, you may want to call that variable $First_Name; it makes it easier for you and anyone else to understand exactly what that variable should hold at any given time. It wouldn't make any sense to put an address or number in a variable called "$First_Name". There are also some automatic (built-in) read-only variables that should be avoided to prevent confusion. $path, $host, or $false should not be used as variables for example. PowerShell has a significant library of help files. Virtually every cmdlet has a help file that will give much more detail on the use, parameters, and structure of the object. Help Set-Variable, for example will give much more details around the Set-Variable cmdlet. For more information and details on PowerShell reserved keywords you can refer to Microsoft's resource documents: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_reserved_words?view=powershell-7.2

If you are having issues with getting the right data out or some error regarding the contents of a variable, it may be that you have assigned your variable a reserved keyword.

2.4 Write-Host

With data stored in a variable we need a way of retrieving it. One of the simplest ways to do that is to simply display the value of the variable in the terminal. This process of displaying the contents of a variable or string is known as `writing out`. There are several ways to write out your data but the simplest one is to write-host.

Figure 2.3 What's a variable?



Write-Host writes its output to the console you're working in. As you can see in the above screenshot the value of variable *$Name* was printed to the PowerShell output of my VS Code IDE window.

2.5 Modifying Strings

When PowerShell interprets a string; it recognizes it as an object. Each object type has a number of actions that can be done to it. These actions are known as methods in the object orientated programming (OOP) languages. What they are and how they work are not important for you to know right now. But each object will have different methods that can be done to them.

A Brief Look at Programming Objects

PowerShell is what's known as an Object Orientated Programming (OOP) Language. Objects in programming terms are a collection of common attributes and methods. Think of any other type of (non-programming object), like a cat. There are things that all cats have in common:

- They have four legs
- They have fur
- They have teeth
- They have claws
- They have a tail

Etc.

These are known as the Attributes of an object because they describe what the object has.

There are also things that you can do to cats:

- You can Feed them
- You can Pet them
- You can Call them
- You can Bathe them

These are known as Methods of an object because they describe what you can do to the object. Typically, however, when we talk about Methods in programming terms, they are described as things an object can do by themselves when we call them.

Like non-programming objects, Objects in PowerShell have things that they have (Attributes) and things that you can do to them (Methods). We will look at some of these Methods to see how we can use them to modify our string.

Keep in mind, just as different non-programming objects have different things that can be done to them; so too do Objects in PowerShell.

For example, a house may have doors you can open, or cats may have fur you can comb, but you cannot comb a door. There are some methods that are available to some objects that are not available to others.

We'll touch more on methods in later chapters, so don't get bogged down in the details. Right now, we're just going to utilize several methods to modify our string and help us do useful things with it.

Note

For anyone curious about methods and attributes of a string you can use the Get-Member cmdlet. To see all of the methods available for the string object type the following code into a PowerShell prompt.

```
"Test"|Get-Method
```

Don't worry if you don't understand what this is doing at this point, we'll explain how to Pipe data to cmdlets in later chapters. For now, just browse through the methods available to the object. We'll touch on a few of these in detail later in this chapter.

2.5.1 Trim

Earlier, we assigned values to two variables:

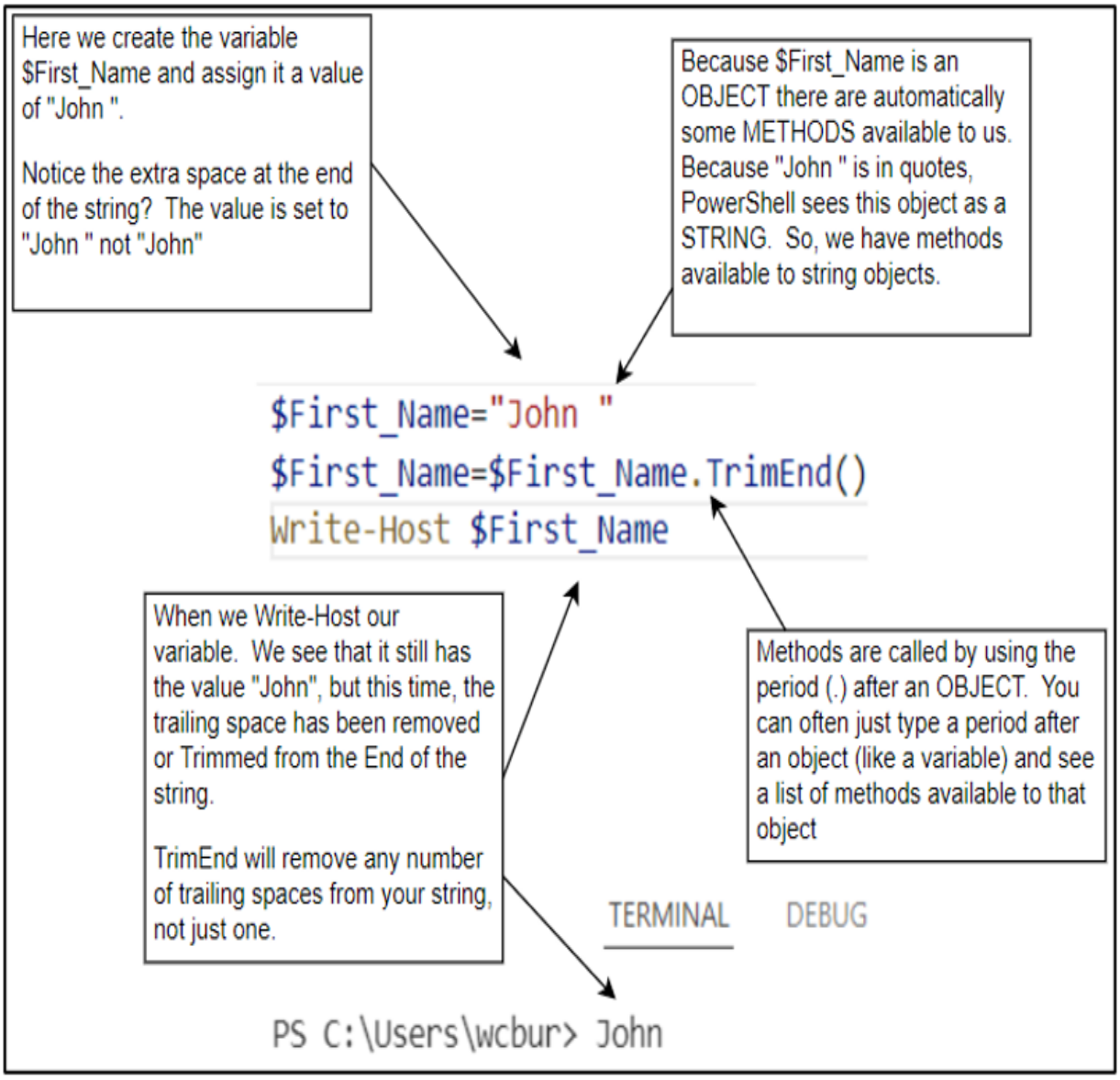
```
$First_Name="John "
$Last_Name=" Doe"
```

Notice how both the `$First_Name` and *$Last_Name* have extra spaces? Because of this, the *$Name* has two extra spaces. It has a space after *$First_Name* and a space before *$Last_Name.* To fix this we can use the *trim* method on the *$First_Name* string.

```
$First_Name="John "
$First_Name=$First_Name.TrimEnd()
Write-Host $First_Name
```

In this case, we use the method *TrimEnd.* This will trim the trailing space or spaces from the end of the string.

Figure 2.4 Using TrimEnd



Notice, however, how difficult it is to see if the space was removed? One quick way to check to see if there is any leading or trailing white space on any of your variables is to put some sort of special character directly before and after the variable in your write-host.

```
$First_Name="John "
$First_Name=$First_Name.TrimEnd()
Write-Host "*$First_Name*"
```

In this case, I have placed a "*" both before and after the variable containing our $First_Name value.

Single versus Double Quotes for Strings

As I have mentioned earlier in this chapter PowerShell recognizes both the single quote (') and the double quote (") as designations for strings. However, PowerShell does not necessarily treat both of these quotes the same. PowerShell treats single quotes as literal strings. This means that PowerShell literally types out exactly what is within the single quotes.

Double quotes, however, are treated as expandable quote. This means that PowerShell will recognize variables within the quotes and instead of literally typing out the text, it will substitute the *value* of the variable instead.

You must always balance your quotes. If you start a string with a single quote, you must end it with a single quote. If you start with a double quote, you must end with a double quote.

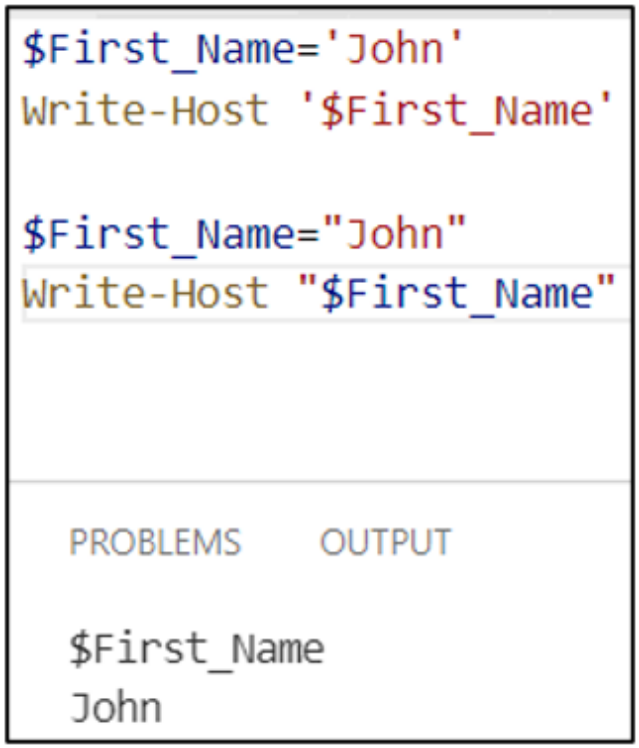
Notice the difference. At first, they seem the same, but PowerShell will interpret them very differently.

```
$First_Name='John'
Write-Host '$First_Name'

$First_Name="John"
Write-Host "$First_Name"
```

PowerShell displays $First_Name (or the literal string typed) when $First_Name is surrounded by single quotes.

But when we use double quotes around "$First_Name" we see PowerShell instead understands we want the value of the variable $First_Name to be returned and writes "John" to our terminal.

Figure 2.5 Single and double quote examples

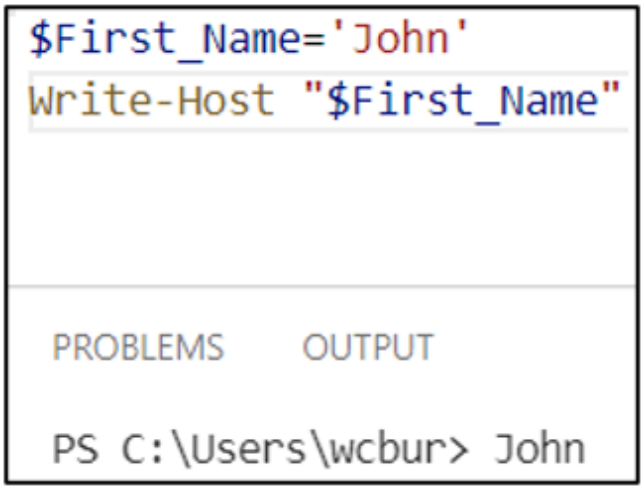


Mixing quotes within your script is fine.

```
$First_Name='John'
Write-Host "$First_Name"
```

This works just fine. However, do not mix quotes within your string.

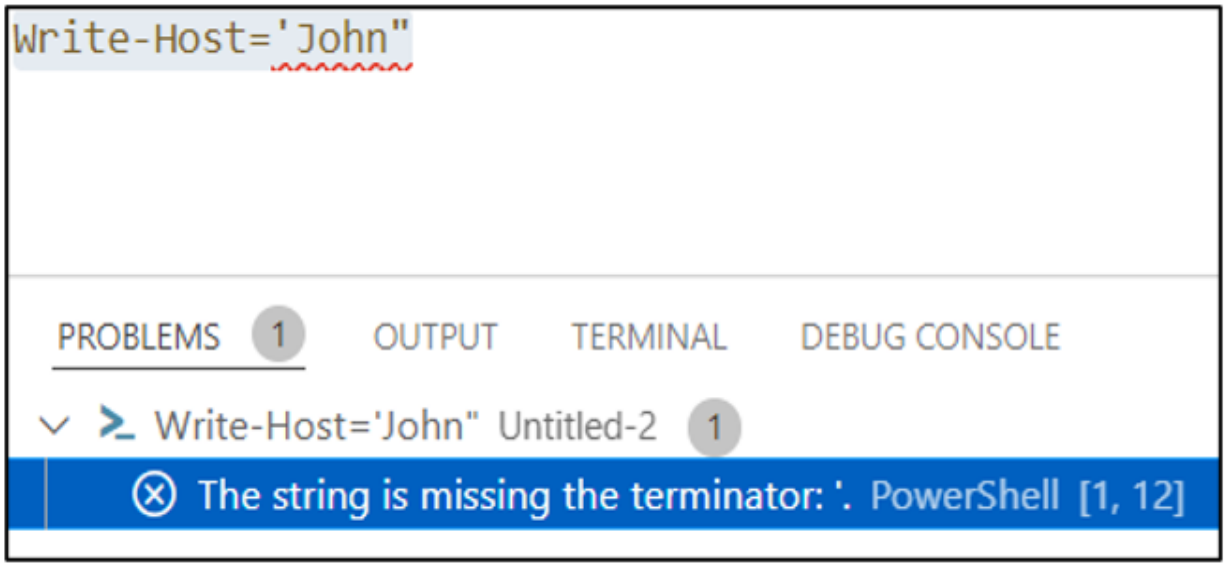
Figure 2.6 Mixed quote example.



```
$First_Name='John"
```

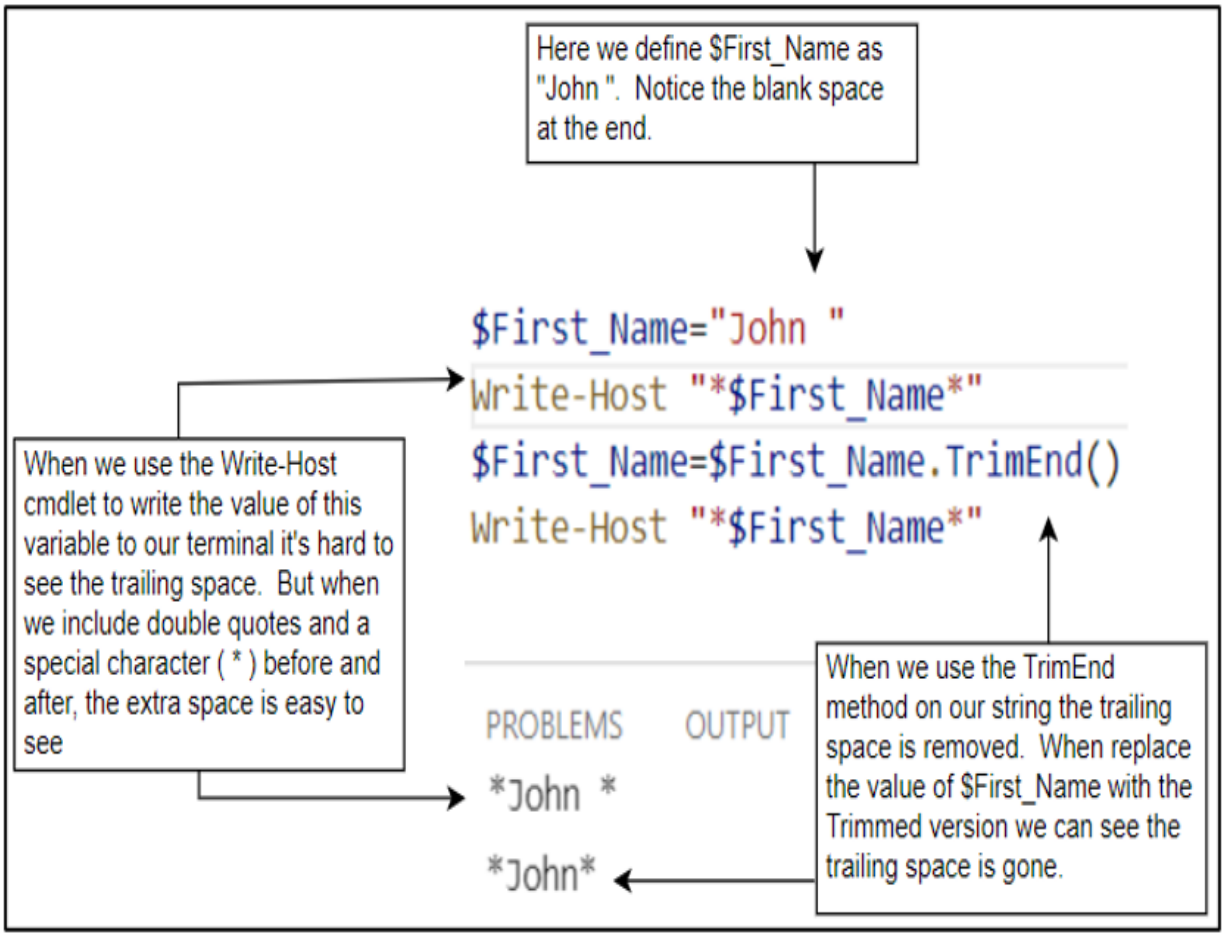
This will produce an error. We'll learn more about errors and how to deal with them in Chapter 4.

Figure 2.7 Mixing quotes within a string causes an error.



Notice in the script below we are using double quotes. Single quotes would give a literal value and the Write-Host would not give us the output we are looking for.

Figure 2.8 Checking for Whitespace

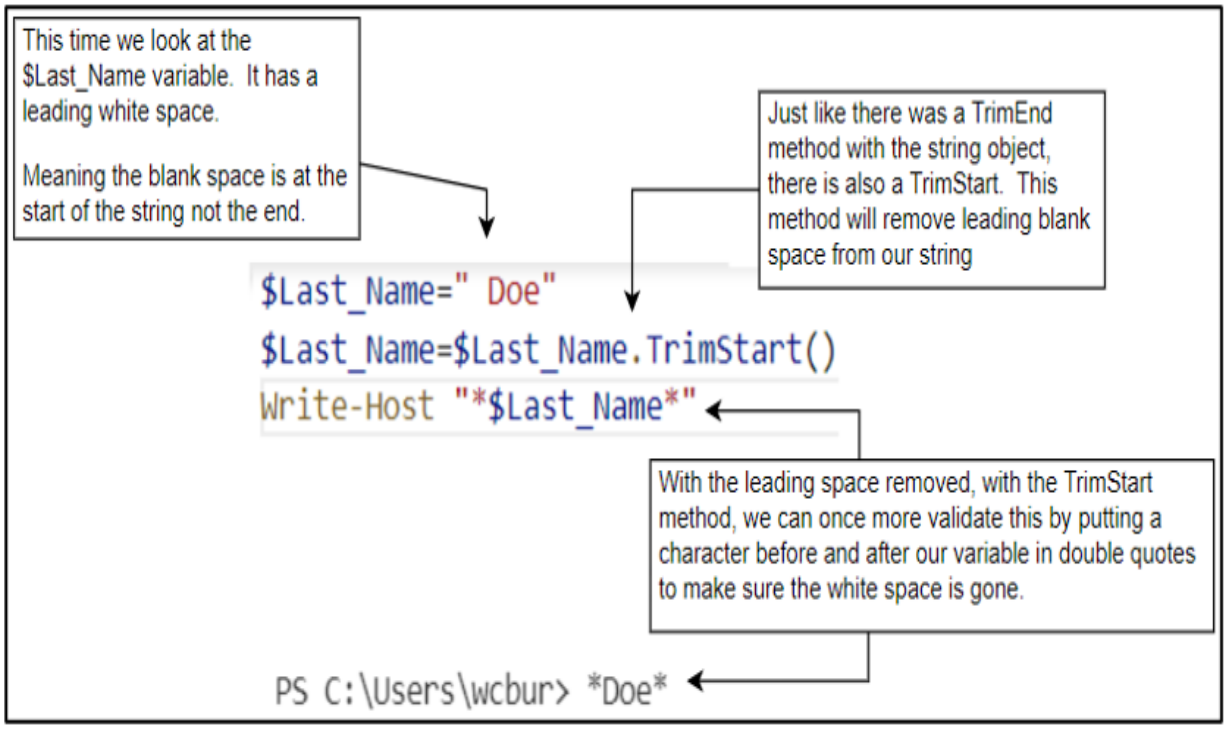


As you can see, the white space has been removed with the *TrimEnd*.

Similarly, the leading space in the *$Last_Name* can be removed with the *TrimStart* Method.

```
$Last_Name= "Doe"
$Last_Name=$Last_Name.TrimStart()
Write-Host "*$Last_Name*"
```

Figure 2.9 Using TrimStart

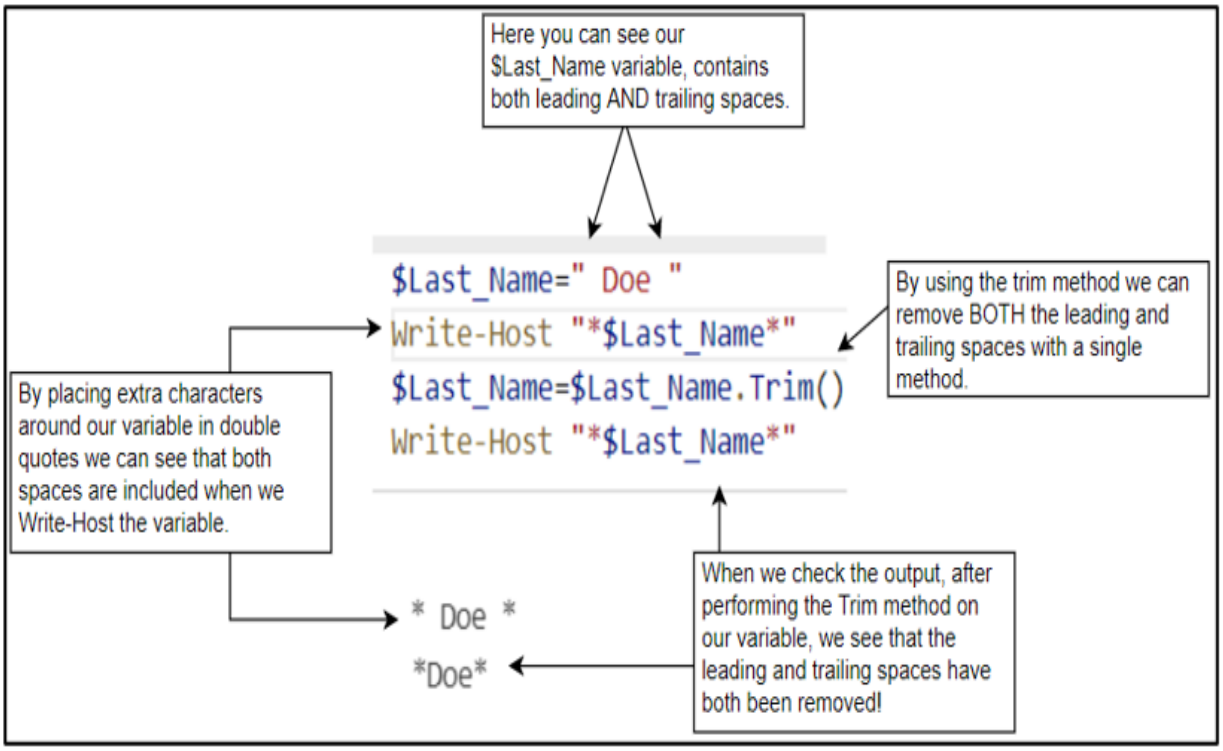


Success! We have removed the leading white space with the *TrimStart.*

If you want to remove all leading and trailing white space from a string OBJECT you can use the *Trim* method.

```
$Last_Name="Doe"
Write-Host "*$Last_Name*"
$Last_Name=$Last_Name.Trim()
Write-Host "*$Last_Name*"
```

Figure 2.10 Using the Trim() Method



Utilizing the *Trim* method, we have removed all leading and trailing spaces in the variable object.

2.5.2 Substring

Substring is a method associated with strings that allows you to select specific ranges of the string. You can select the first character, the middle characters, or any series of characters from the string. In our case, our email address started with the first character of the first name followed by the last name. You can select the first character by using the *Substring* method.

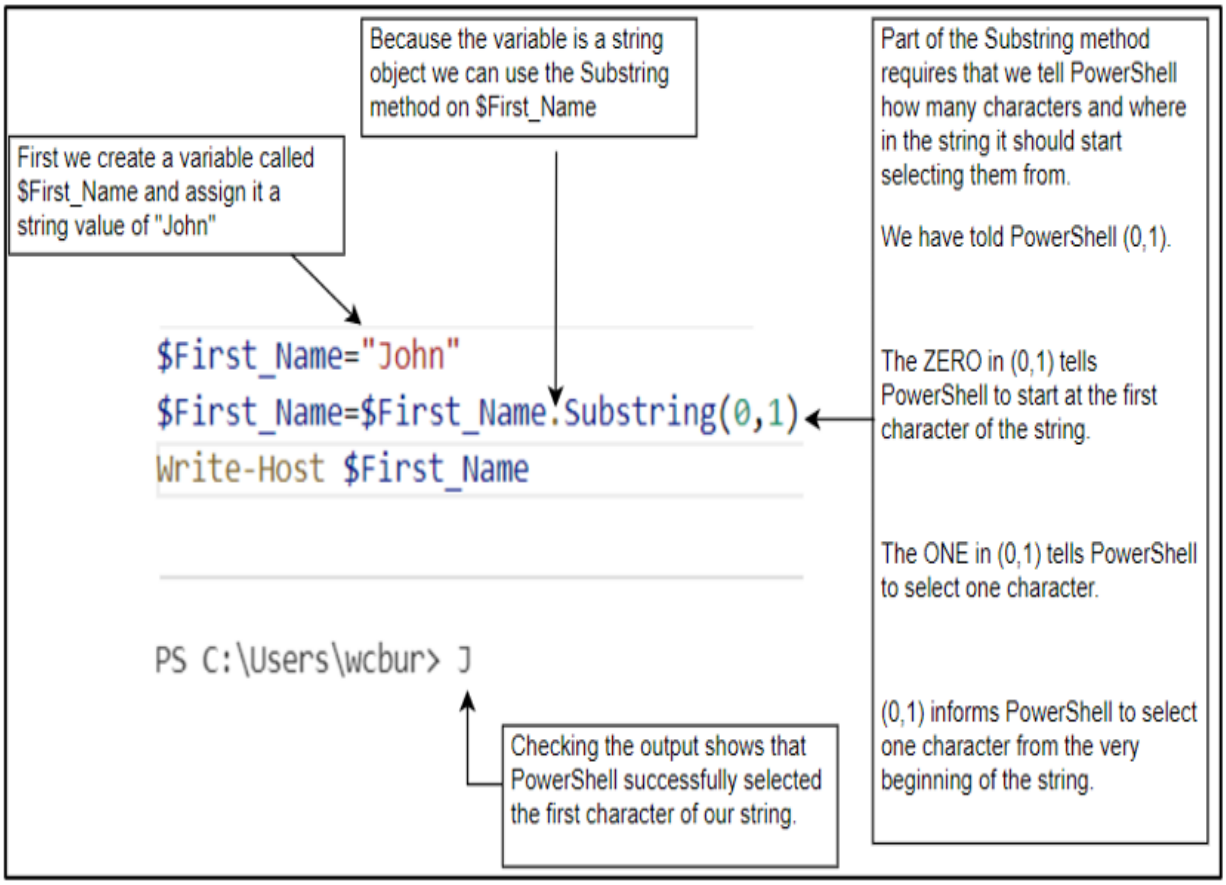
Using the *Substring* method we will pull out the first letter and only the first letter of the string and save it as $First_Name.

```
$First_Name="John"
$First_Name=$First_Name.Substring(0,1)
Write-Host $First_Name
```

What do the parenthesis mean in the Substring Method? *Substring* uses two characters to denote how much of the string is captured. The first number in the parenthesis denotes the starting point of the string where PowerShell

should start selecting characters. The second number is the amount of characters we want the *Substring* method to return.

Figure 2.11 Using the Substring Method

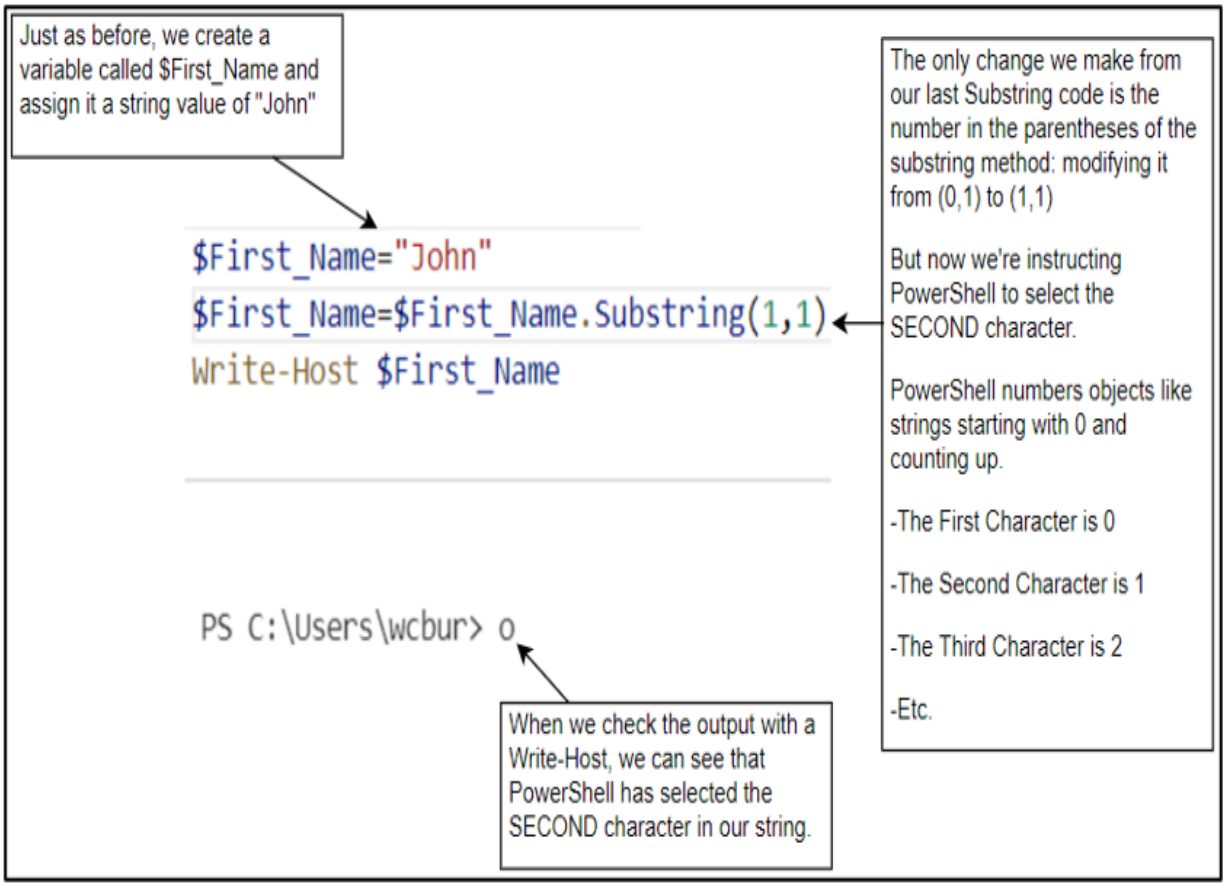


You might be curious as to why, if the first number represents the starting position of the string we want to capture, we are using 0 and not 1.

This is because it's common in programming languages to have the first number in any counting sequence start with 0 and not 1.

If we change the starting position to 1, you'll notice we grab the second character in the string and not the first.

Figure 2.12 Selecting a different character using SubString



Keep this numbering in mind. We'll get into the specifics in chapter 5 when talking about arrays.

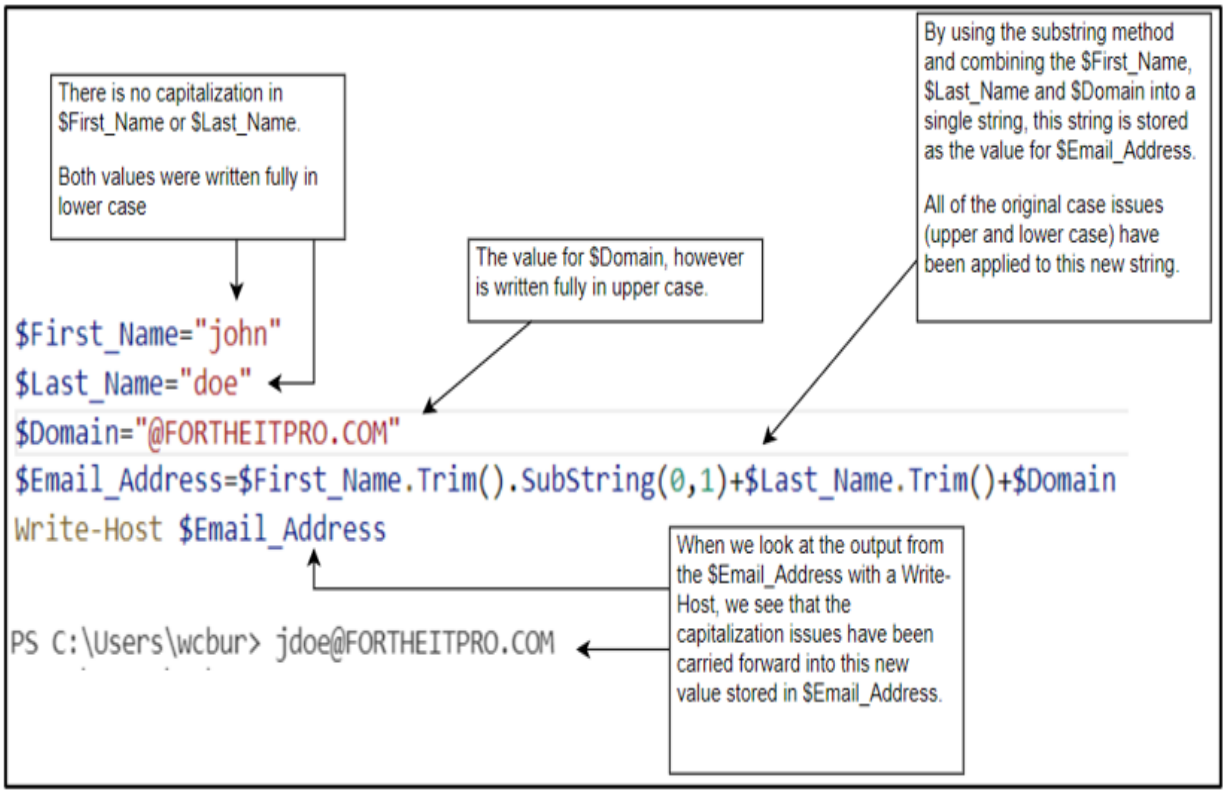
2.5.3 ToUpper

When using this script to help you generate email addresses, you are probably copying and pasting the names into the *$First_Name* and *$Last_Name* fields of the script and letting PowerShell do the rest of the work. The idea of a script, after all, is to let the computer do the work for you. But what if you are pulling these names from a spreadsheet or some other source and the names are written in a different case?

```
$First_Name= "john"
$Last_Name="doe"
$Domain="@FORTHEITPRO.COM"
$Email_Address=$First_Name.SubString(0,1)+$Last_Name+$Domain
Write-Host $Email_Address
```

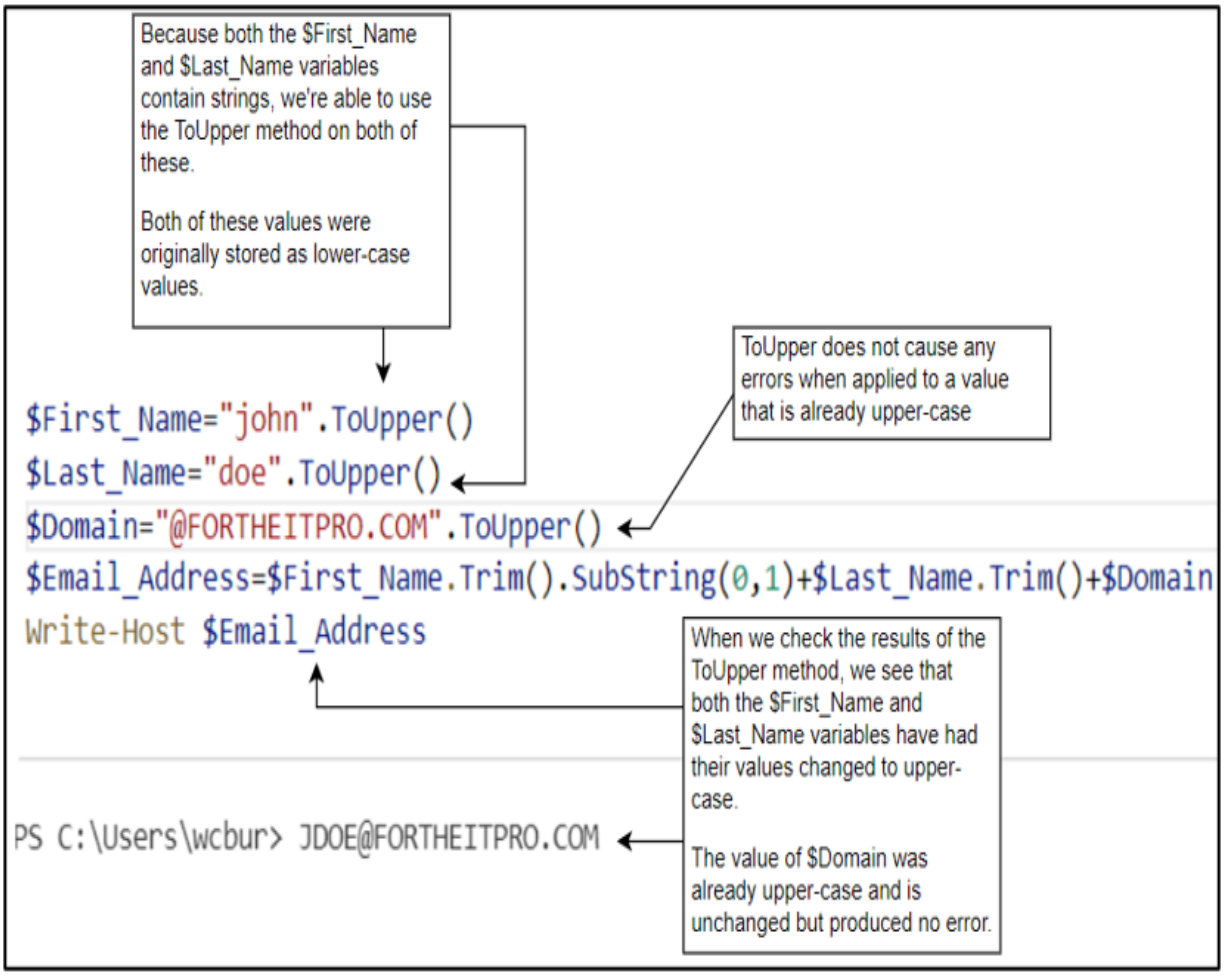
This code still works! But it looks ugly and unprofessional especially when compared to several email addresses with the correct case formatting.

Figure 2.13 Mixed Case Examples



You can fix this by converting all of your strings to a case of your choosing. *ToUpper* is a method for strings that will replace every character of the string with the identical Upper-Case letter.

Figure 2.14 Using ToUpper Method to resolve mixed cases

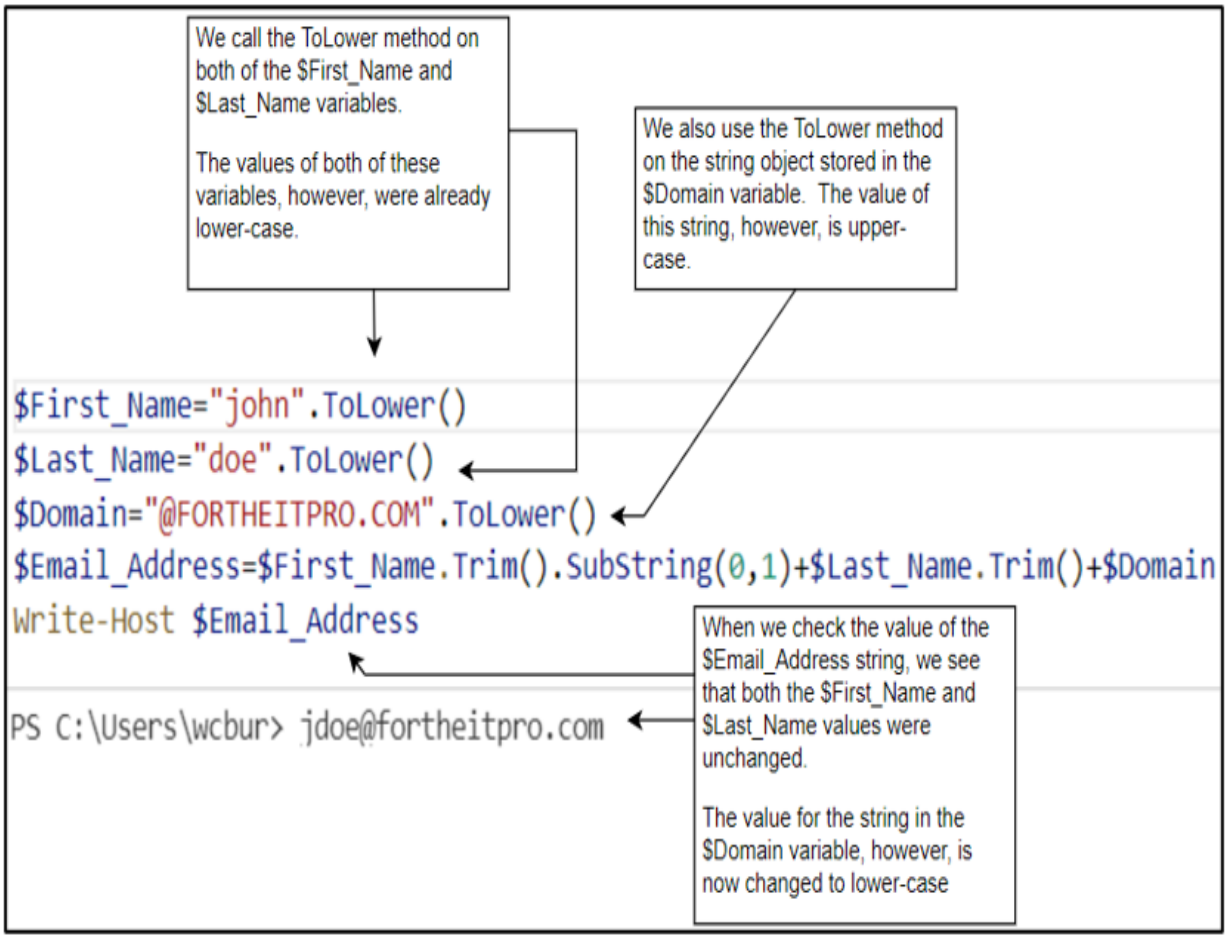


This will make every character in all of the strings Upper Case for more universal formatting.

2.5.4 ToLower

Much like *ToUpper, ToLower* modifies all of the characters in the strings to lower case version of the characters.

Figure 2.15 Using ToLower method to resolve mixed cases.



PowerShell 6.0x onward, PowerShell gains the ability to work across platforms. Operating Systems like Windows and Mac are case insensitive; however, Linux operating systems are not. When running PowerShell on non-Windows Operating Systems the case of your input and output become important. Utilizing ToUpper, ToLower and Substrings should be able to case any of your variables correctly regardless of the original case of the variable.

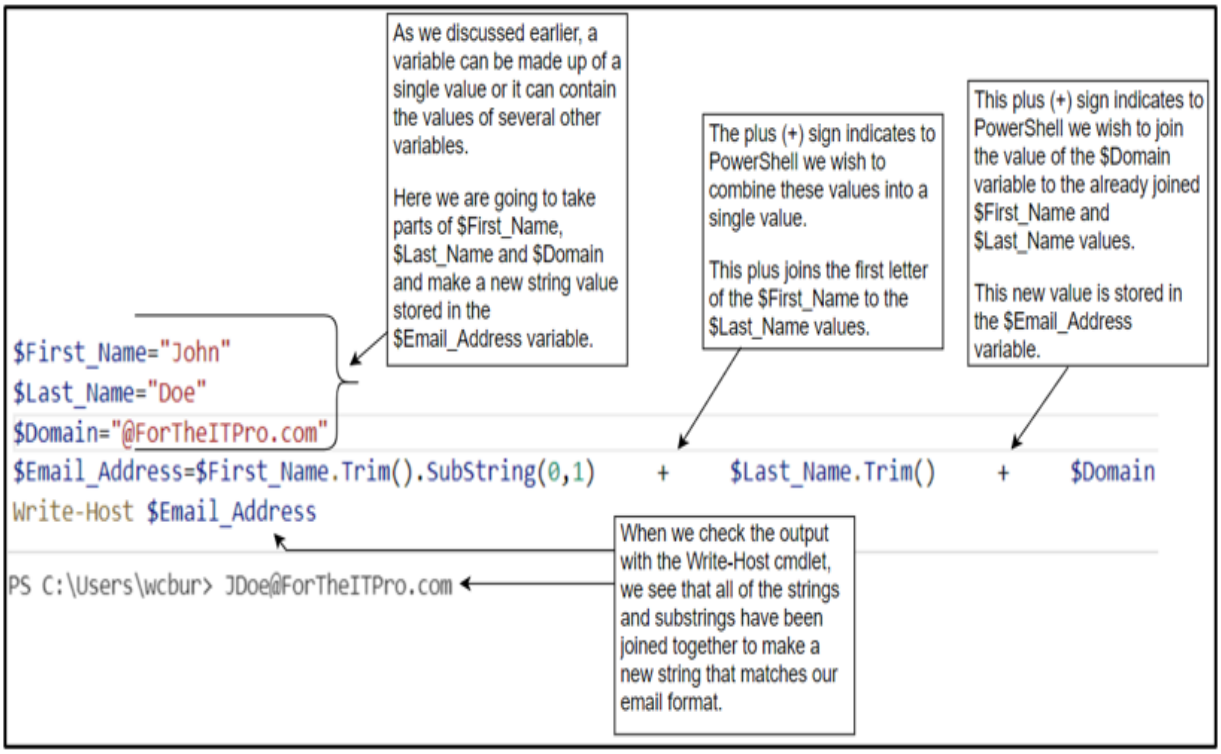
2.5.5 Concatenation

Often you want to take different parts of stings and combine them into a single string. In the email address example above, we have a *$First_Name, $Last_Name* and a *$Domain* that we are combining into a singular variable called *$Email_Address*. We can accomplish this with string *concatenation.*

Like many things in PowerShell there are many ways to do this. We have done it in this example with the use of a plus (+) signs. This tells PowerShell that we want to combine all of these things into a single item.

```
$First_Name="John"
$Last_Name="Doe"
$Domain="@ForTheITPro.com"
$Email_Address=$First_Name.SubString(0,1)  +  $Last_Name  +
$Domain
Write-Host $Email_Address
```

Figure 2.16 Using Concatenation



Because we haven't put any spaces in quotes or between cmdlets, PowerShell ignores white space in this code. So, we can space this code out a little bit to highlight the joiners and indicators of this concatenated statement.

2.6 Try this too!

Utilizing the *Substring* commands, you can modify exactly what type of information gets cut and *concatenated* to make your emails. Maybe, for example, your company doesn't use the "first initial, last name" format for its email system. Maybe instead, they pull the first two letters of the person's first name and the first two letters of their last name *concatenated* with their domain. By playing with *Substring* you can make this modification easily.

```
$First_Name="John"
$Last_Name="Doe"
$Domain="@ForTheITPro.com"
$Email_Address=$First_Name.Substring(0,2)+$Last_Name.Substring(0
,2)+$Domain
Write-Host $Email_Address
```

Congratulations, to steal a quote from ObiWan Kenobi, "*You've taken your first step into a larger world*.". Embracing the power and flexibility that scripting language brings to your PowerShell skillset will be game changing.

In later chapters we'll explore ways to make it so PowerShell reads directly from our input source eliminating the need to copy and paste manually!

2.7 Summary

- PowerShell stores information in an OBJECT called a Variable.
- Variables can hold different types of data.
- One of these data types is known as a STRING, defined by surrounding the string with quotations: either single quotes (') or double quotes (").
- STRINGS have different METHODS that can modify them in useful ways
- We can TrimStart, TrimEnd or Trim STRINGS to remove blank spaces from the start or end of our STRINGS or both.
- We can select any part of our STRING with SubString.
- We can change the whole case of the STRING with ToUpper or ToLower.
- We can combine several variables and text into another STRING by concatinating it.
- We can check the value of any STRING by using Write-Host.

3 Create a user (the easy way)

This chapter covers:

- Using Read-Host for user supplied data
- Creating Menus with Read-Host and Write-Host
- PowerShell pipe mechanism
- Importing Modules
- Active Directory cmdlets New-ADUser and Get-ADUser
- Filtering AD Users with Where-Object
- Introduction to Race Conditions.
- Logging command history quickly and easily.

Adding users is a staple of a System Administrator's world. The fact is, if there are not any users, there's not really much point in having a system. Many systems have hundreds or even thousands of users. You can bet that nearly every one of them was created by a System Administrator. You can also bet that most of them were likely created by hand.

Most System Administrators view creating a user account as a rather trivial task, and by and large it is. But there are several required fields that make copy and pasting more than a little bit of a pain. Many companies, to save on onboarding costs typically have whole newhire classes. As such, it's often that a System Administrator is not adding a single account, but often dozens at a time.

This usually involves:

- Opening Active Directory Users and Computers
- Finding a general user account (one without Administrator Rights or special permissions)
- Right Clicking and selecting Copy
- Entering a First Name
- Entering a Last Name
- Entering a Full Name

- Entering a unique Username (you remembered to look that up before clicking copy right?)
- Selecting Next
- Clicking the Account is Disabled (Or entering Generic Password)
- Selecting Next
- Selecting Finish

3.1 Project Code

What if, instead you simply had to open your New User script

- Press F5
- Enter a First Name
- Enter a Last Name

In addition to being significantly fewer keystrokes and clicking what if this script could also keep a record of every User created? Piqued your interest yet? Let's take a look.

Warning

This code is frequently broken to span multiple lines. This is done at the pipe (|) operator. We'll learn more about the pipeline mechanism later in this chapter! This will run just fine in PowerShell 7.x. However, this may prove problematic with older versions of PowerShell. If you plan on running this code utilizing PowerShell 5.1, you can fix this by moving every line that starts with a pipe (|) to the end of the line before it.

First, we'll look at the code that works in both the legacy Windows PowerShell 5.1 as well as the newer PowerShell 7.x:

Figure 3.1 Works in PowerShell 7.x and PowerShell 5.1

```
Get-ADUser -filter *|Where-Object {$_.SamAccountName -eq $User_Name}|Tee-Object $Log_File -Append
```

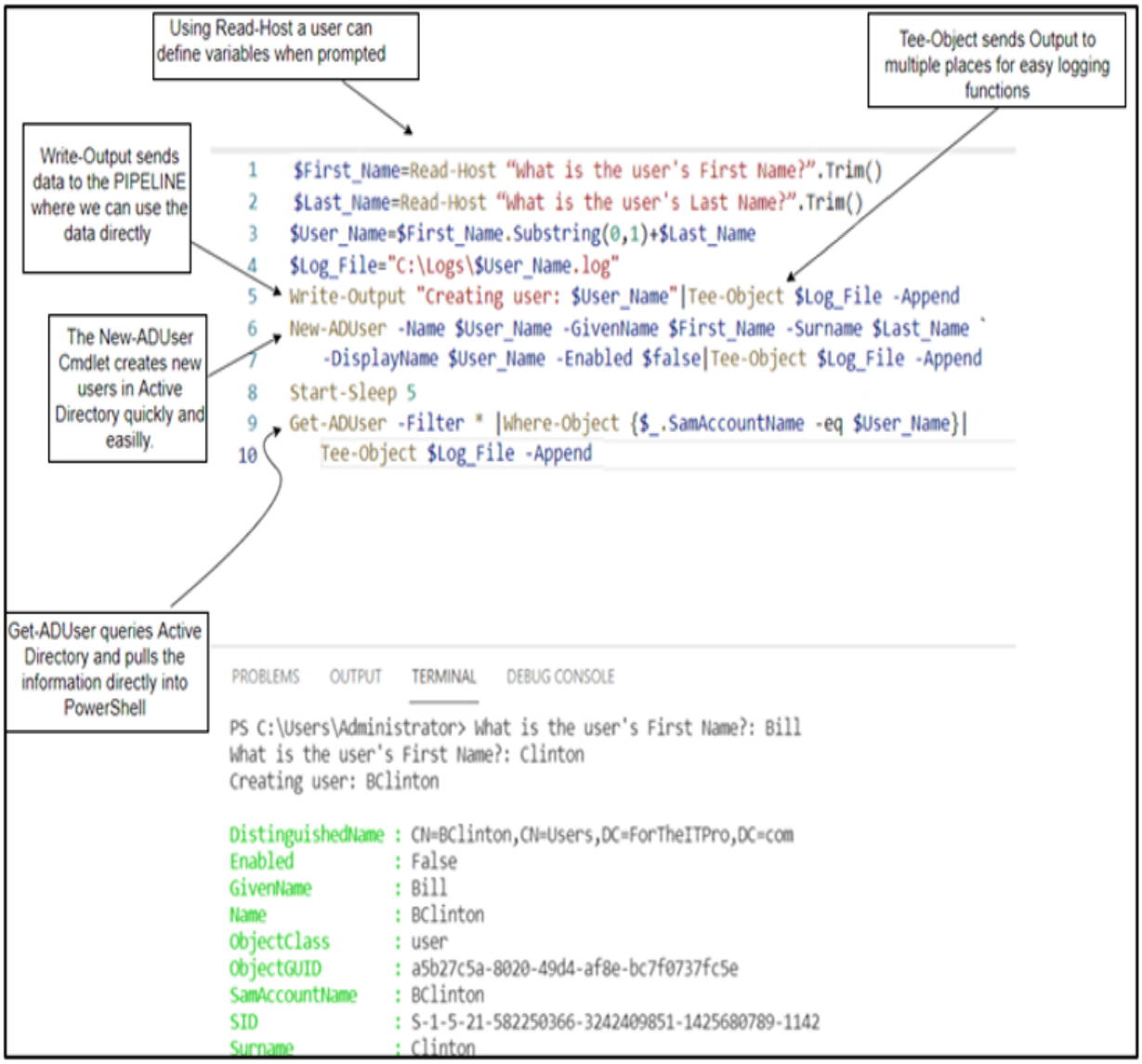
Now, notice the difference in Figure 3.2, below; this script operates perfectly in PowerShell 7.1, but breaks in the legacy Windows PowerShell 5.1 and lower.

This is shown in figure 3.3 line 9 and 10. The line currently reads:

Figure 3.2 Works in PowerShell 7.x but not PowerShell 5.1

```
Get-ADUser -filter *|Where-Object {$_.SamAccountName -eq $User_Name}
     |Tee-Object $Log_File -Append
```

Figure 3.3 High Level view of the user creation script




```
$First_Name=Read-Host "What is the user’s First Name?".Trim()
$Last_Name=Read-Host "What is the user’s Last Name?".Trim()
$User_Name=$First_Name.SubString(0,1)+$Last_Name
$Log_File="C:\Logs\$User_Name.log"
Write-Output "Creating user: $User_Name"
    |Tee-Object $Log_File -Append
New-ADUser -Name $User_Name -GivenName $First_Name -Surname
$Last_Name `
    -DisplayName $User_Name -SamAccountName $User_Name -Enabled
$false
    |Tee-Object $Log_File -Append
Start-Sleep 5
Get-ADUser -filter *|Where-Object {$_.SamAccountName -eq
$User_Name}
    |Tee-Object $Log_File -Append
```

3.1.1 What does it do?

- Asks the operator for the user's First Name and takes the input and stores it as the variable *$First_Name.* It also removes any whitespace from the entry, through use of the Trim method learned in chapter 2, to prevent whitespace from becoming part of the Username.
- Asks the operator for the user's Last Name and takes the input and stores it as the variable *$Last_Name*. Like before, it also removes any whitespace to prevent whitespace from becoming part of the Username.
- It `Substrings` the first character of the user's First Name and Concatenates it with the user's last name storing it as a variable called *$User_Name.*
- Creates a variable called *$Log_File* and assigns it the value of "C:\Logs\ concatenated with the newly created *$User_Name* and appended a log file.
    - If the *$User_Name* were JDoe the *$Log_File* would be C:\Logs\JDoe.log.
- It then displays to the operator the message "Creating user:" concatenated with the newly created *$User_Name.* It sends this output to both the operator screen and the log file at the same time using a new PowerShell cmdlet called "Tee-Object". We'll learn more about the Tee-Object cmdlet later in this chapter!
- It then creates the ACTIVE DIRECTORY user for the *$User_Name* and disables it. It also sends this information to the log file APPENDING this new information to the log (so as to not overwrite the whole file).
- The script then simply WAITS for five seconds.
- Finally, it queries ACTIVE DIRECTORY looking for the newly created account and sending the results to the log file appending the results to the log file to confirm that the account is now able to be found in ACTIVE DIRECTORY.

Note

In order to use this code in your environment you will need access to an Active Directory Domain Controller. These PowerShell cmdlets come

native to the Active Directory Features you install on your Domain Controller.

To utilize these Active Directory cmdlets without logging into your Domain Controller (remote or console) you can install the Microsoft Remote Server Administration Tools (RSAT). Directions on where to download and how to install this can be found in the README.md included with the Code for Chapter 3.

3.2 Comments

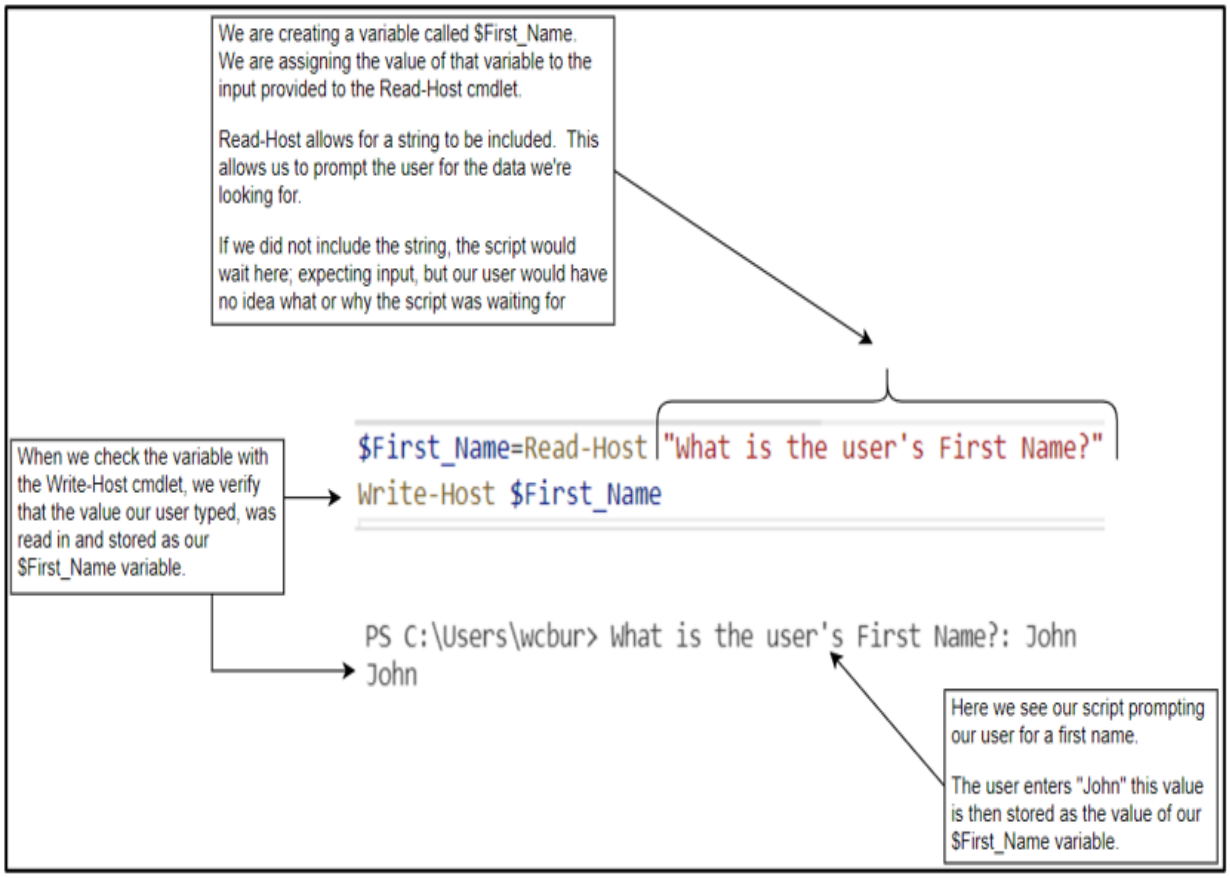
Frequently, when we are scripting or writing any kind of code, we want a way to remind ourselves, or anyone else reading our code what we are doing and why. The easiest way to do this is with a comment. Comments are ways that we can insert an explanation or annotation into our script without it being executed.

There are two ways to input comments into PowerShell scripts:

1. Single line comments
2. Multi-line comments

A single line comment is initiated with the pound or hash sign (#). Anything to the left of the hash is generally ignored by PowerShell.

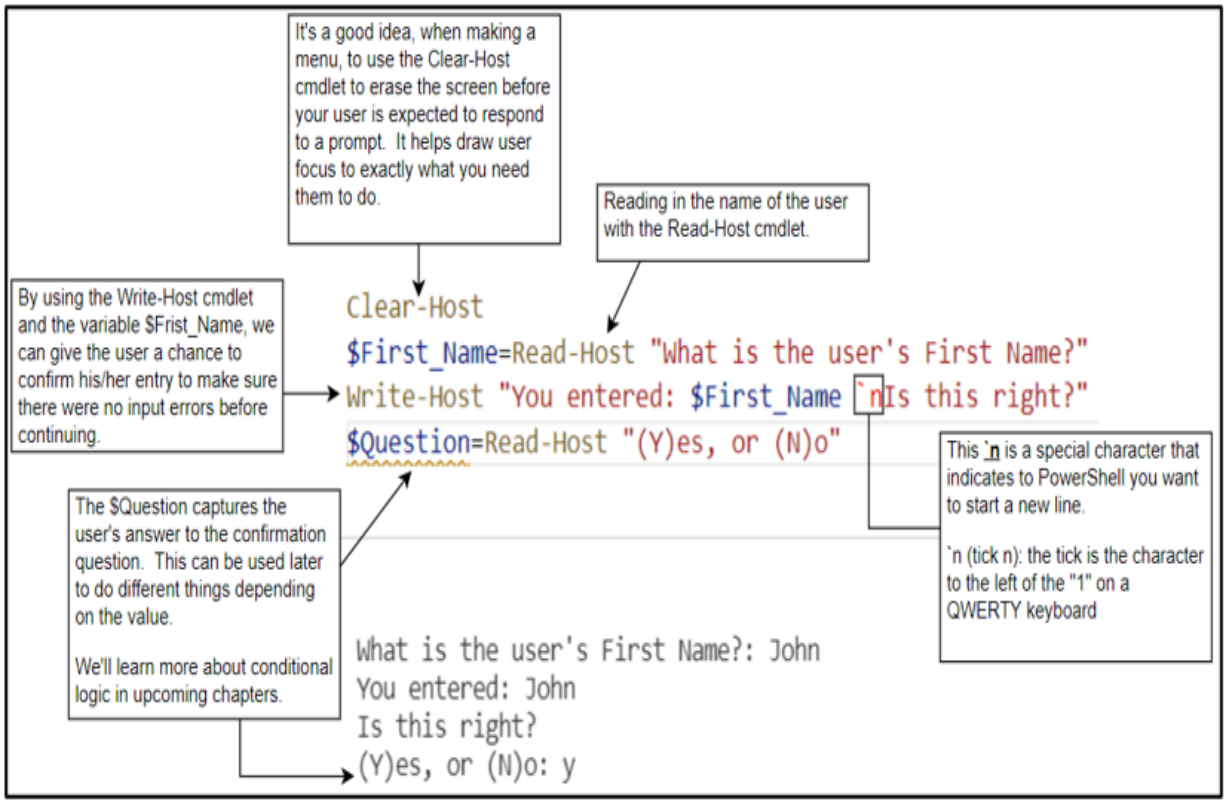
Figure 3.4 Single line comment example.



Multi-line comments begin with a less-than and a hash (<#) and end with a hash and a greater-than (#>)

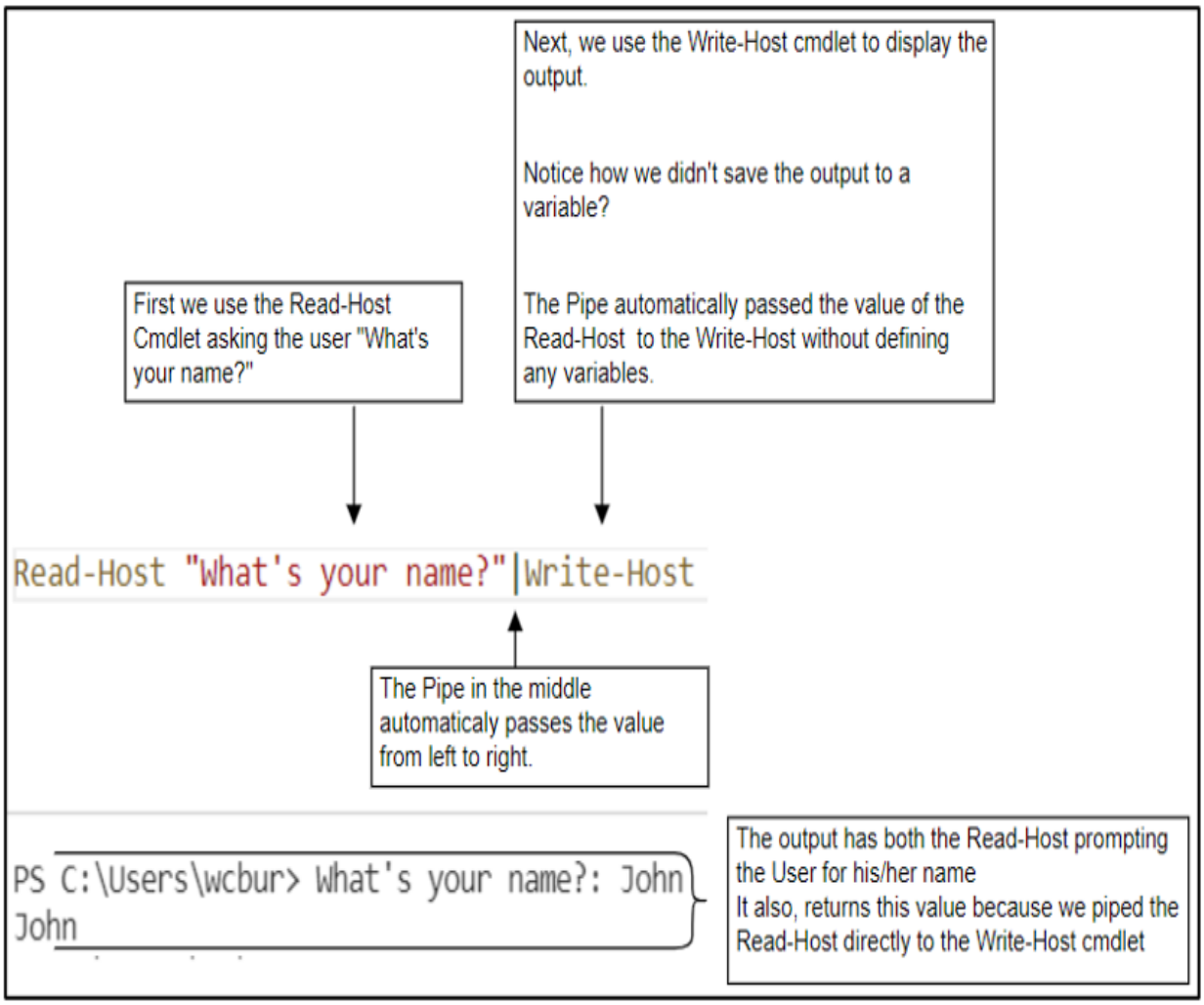
Everything within these characters is treated as a comment.

Figure 3.5 Multi-line Comment example



In this chapter and continuing on, we include several lines of comments at the header of our script that explain to anyone using our script, what it does, how it works and examples of how to use it. PowerShell ignores all of these comments but it makes it much easier for anyone trying to read or use our script to understand how to do so right away with a multi-line comment.

3.6 Code comments example.



3.3 Input/Output

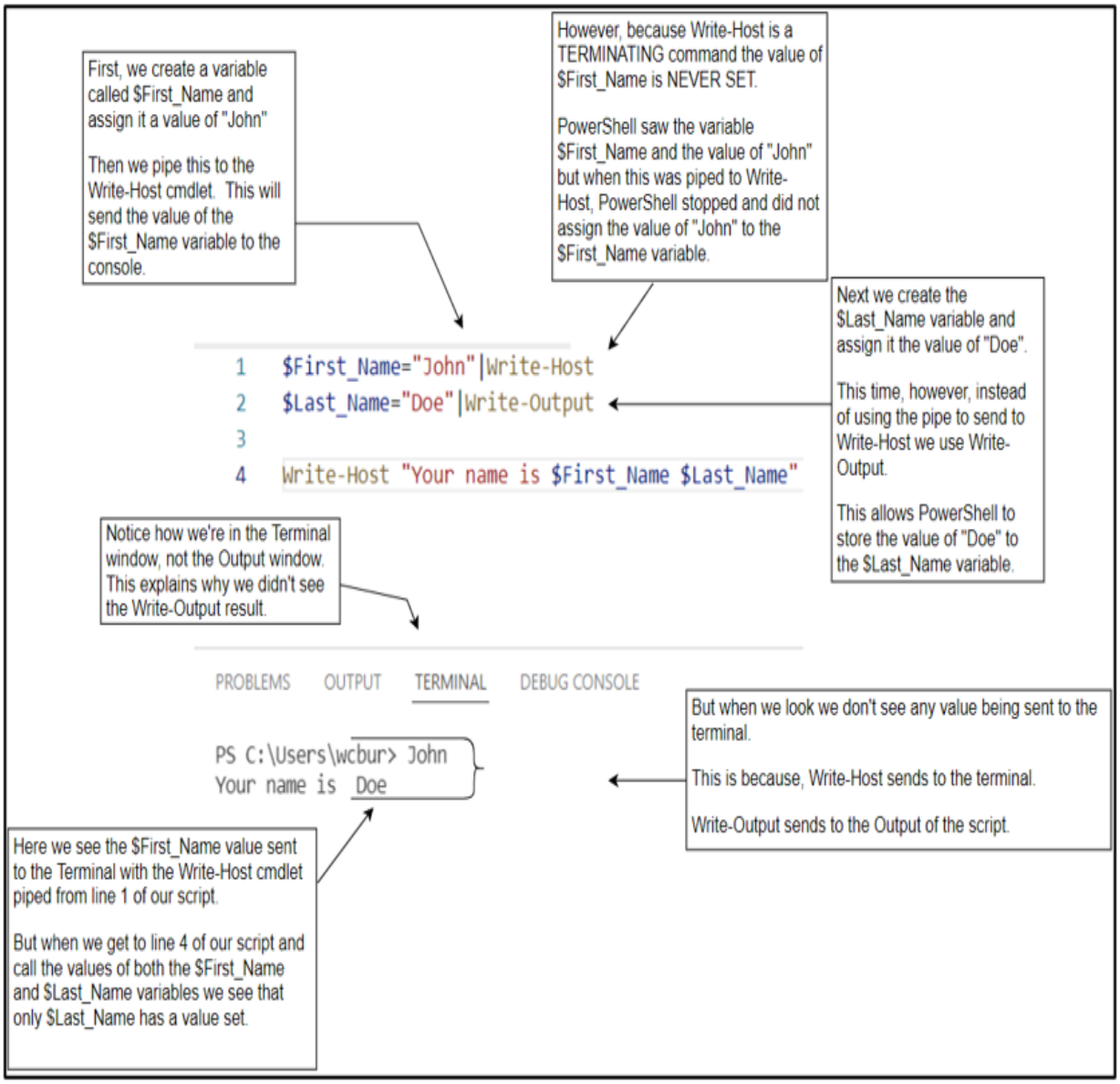
In the last Tiny PowerShell project we hard coded all of our variables. This can quickly become tiresome and dangerous having to re-write, re-save and re-execute your code with every minor variable change. Instead, what if PowerShell asked us the information at runtime and stored this information in a variable for us automatically?

3.3.1 Read-Host

In the last Tiny PowerShell project we used the cmdlet `Write-Host`. This took our data and wrote it to the console for the operator to read. `Read-Host` does the opposite. The console prompts the operator for information. When

we use a variable combined with a `Read-Host`, we can then store this data for use later in our scripts.

Figure 3.7 A dissection of the Read-Host Cmdlet




```
$First_Name=Read-Host "Enter the user's First Name"
Write-Host $First_Name
```

The string in quotes behind the `Read-Host` is not required. The script will execute just fine without it. But you leave the operator of your script clueless as to what information they are being asked to store. It's a good

idea to give your operator some clue about the what the data will be used for upfront. Without the string at the end the operator simply sees a blinking cursor and the script may hang entirely as the operator is not aware that he/she has anything they need to do at this point.

3.3.2 Menus

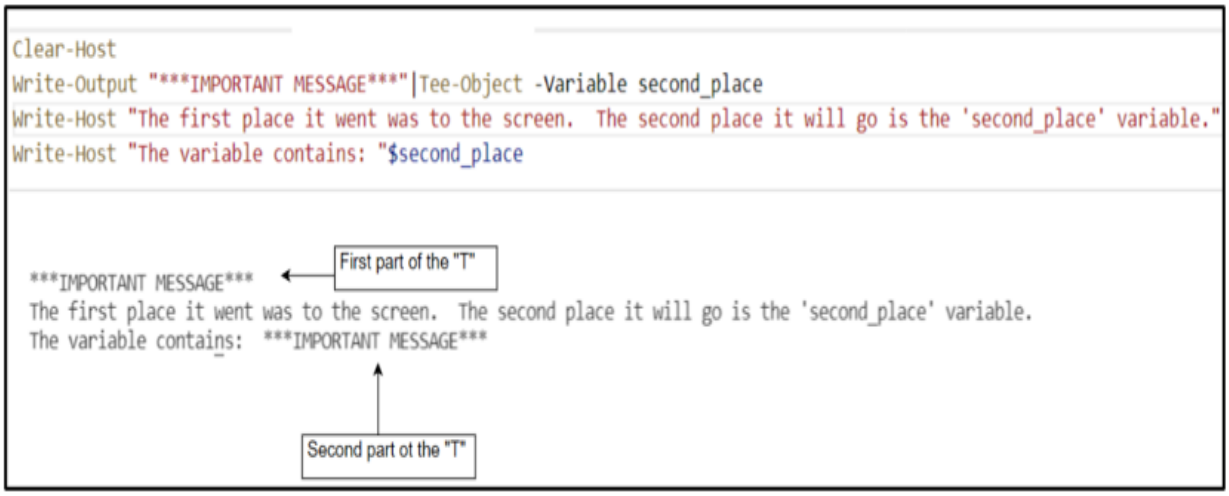
With the use of both Read-Host and Write-Host statements it's easy for your operator to utilize menus or flags that can be used to help make more complex decisions in your code.

```
Clear-Host
$First_Name=Read-Host "What is the user's First Name?"
Write-Host "You entered: $First_Name `n Is that right?"
$Question=Read-Host "(Y)es, or (N)o"
```

Note

In Visual Studio Code, when you type the code above as shown, you will see a red squigly line under the variable called $Question. It may say "The Variable 'Question' is assigned and never used." This is an expected error and can safely be ignored.

Figure 3.8 A simple menu using Read-Host, Write-Host



3.3.3 Output and Pipes

The ability to write messages to the operator of our scripts is very important. There are multiple ways to do this. We have already covered `Write-Host`. Now, we will explore `Write-Output`.

The commands are similar in many ways. Both commands type a message on the screen for the operator to see.

```
Write-Host "Message A"
Write-Output "Message B"
```

These commands do nearly the same thing. Both commands display a message to the console for the operator of your script to read.

To understand the difference, we first need to understand PowerShell's *Pipeline Mechanic*.

3.3.4 Pipe

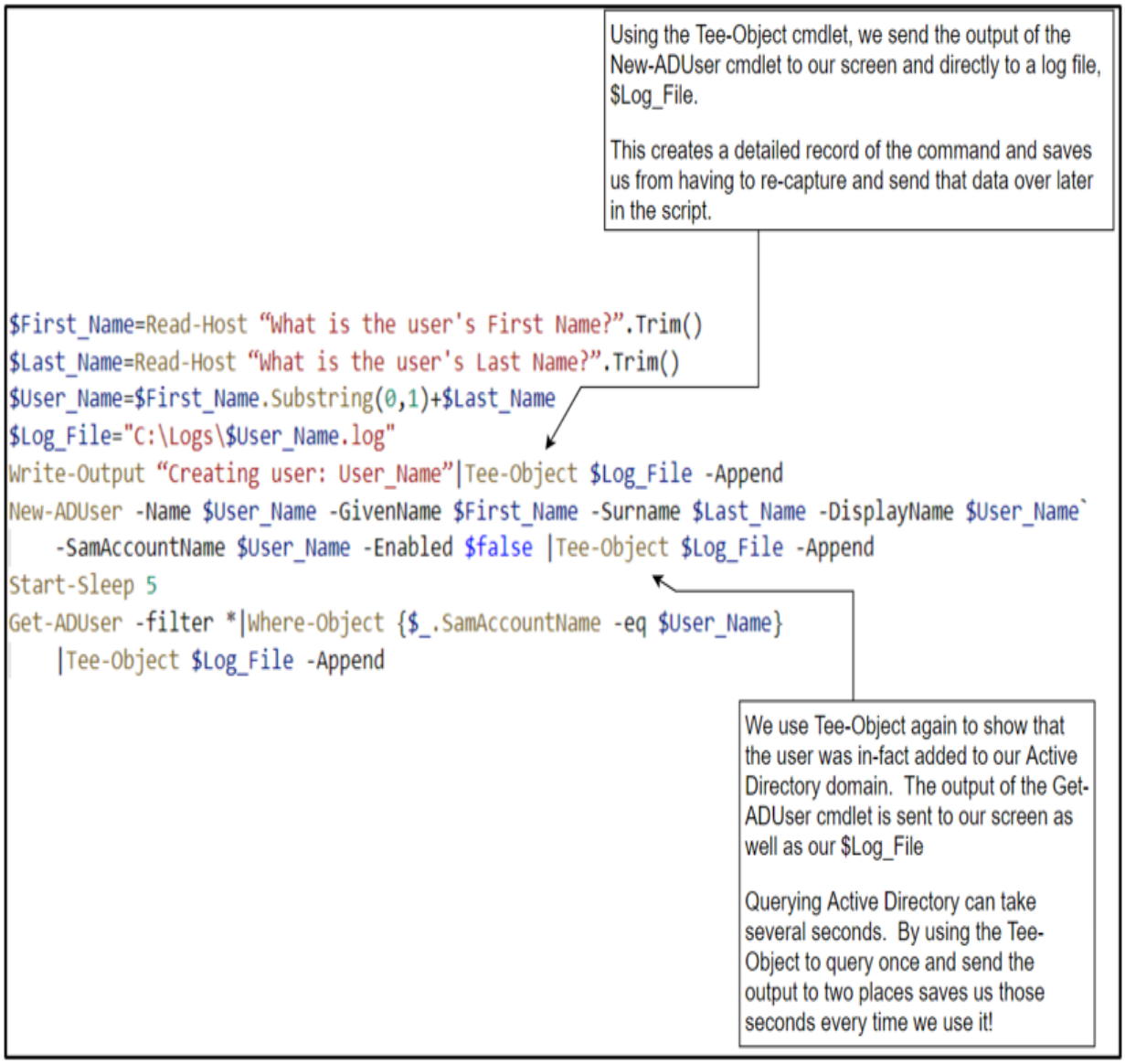
PowerShell allows us to send the output of one command as the input for another command through the use of the pipe operator (|) character. Let's look at the pipeline mechanic in practice utilizing the Read-Host and Write-Host cmdlets. Before, we used the Read-Host cmdlet to input a value into a variable we created, we could then validate this input by outputting the value of the $Name variable to the console with the Write-Host cmdlet.

```
$Name=Read-Host "What's your Name?"
Write-Host $Name
```

Utilizing the Pipeline Mechanic, we can skip the variable entirely and simply pipe the value the user inputs from the Read-Host directly into the console with the Write-Host.

```
Read-Host "What's your name?"|Write-Host
```

Figure 3.9 A breakdown of PowerShell's pipeline mechanic.



Notice how in this code we didn't store anything as a variable. We simply took the output of the first command (`Read-Host`) and piped it directly to the output of the second command the (`Write-Host`).

3.3.5 Write-Output

Now that we're familiar with the *pipeline mechanism* in Windows PowerShell, we can illustrate the difference between the `Write-Host` and `Write-Output` cmdlets.

`Write-Host` cannot store information as a variable. It sends its output directly to the console. Think of it as a message box that you display to your user. Once you display the message it's gone.

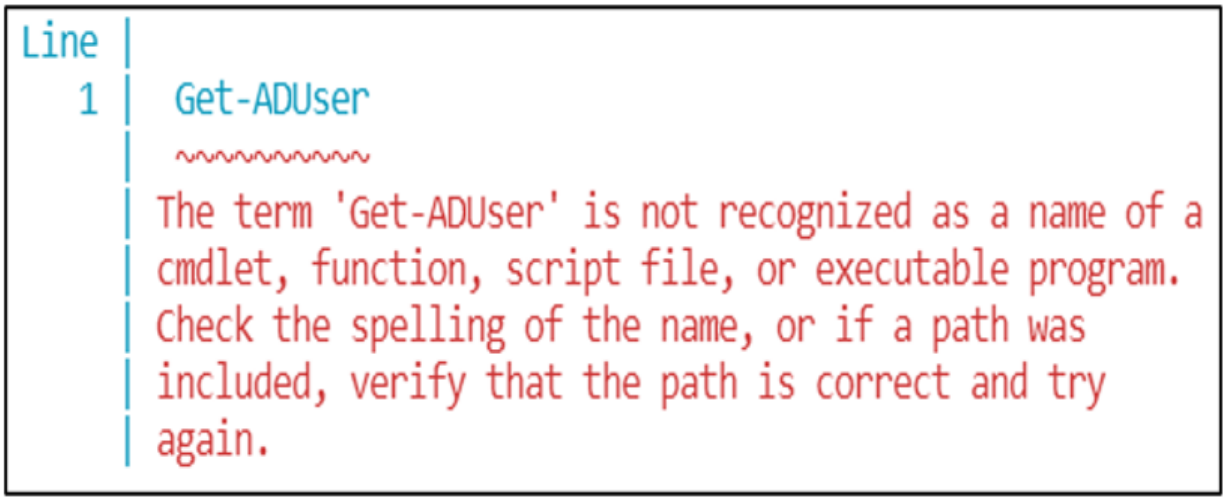
`Write-Output`, however, can store information inside the variable. You can feed output to your user and then utilize that same output in another function (say like a log) without having to assign the data to a variable re-capture or re-create it.

Note

The following code is not something you would normally do in PowerShell, but for demonstration purposes, it will illustrate the difference between the Write-Host and Write-Output. You will quickly see more applicable uses of the Write-Output cmdlet as our PowerShell skillset grows.

```
$First_Name="John"|Write-Host
$Last_Name="Doe"|Write-Output
Write-Host "Your name is $First_Name $Last_Name"
```

Figure 3.10 Differences between Write-Host and Write-Output in action



As you can see, when we used the `Write-Host` the message was output to the operator console, but it was not actually stored as a variable. The *$Last_Name* was stored as a variable and later used by our script as we intended.

3.3.6 Tee-Object

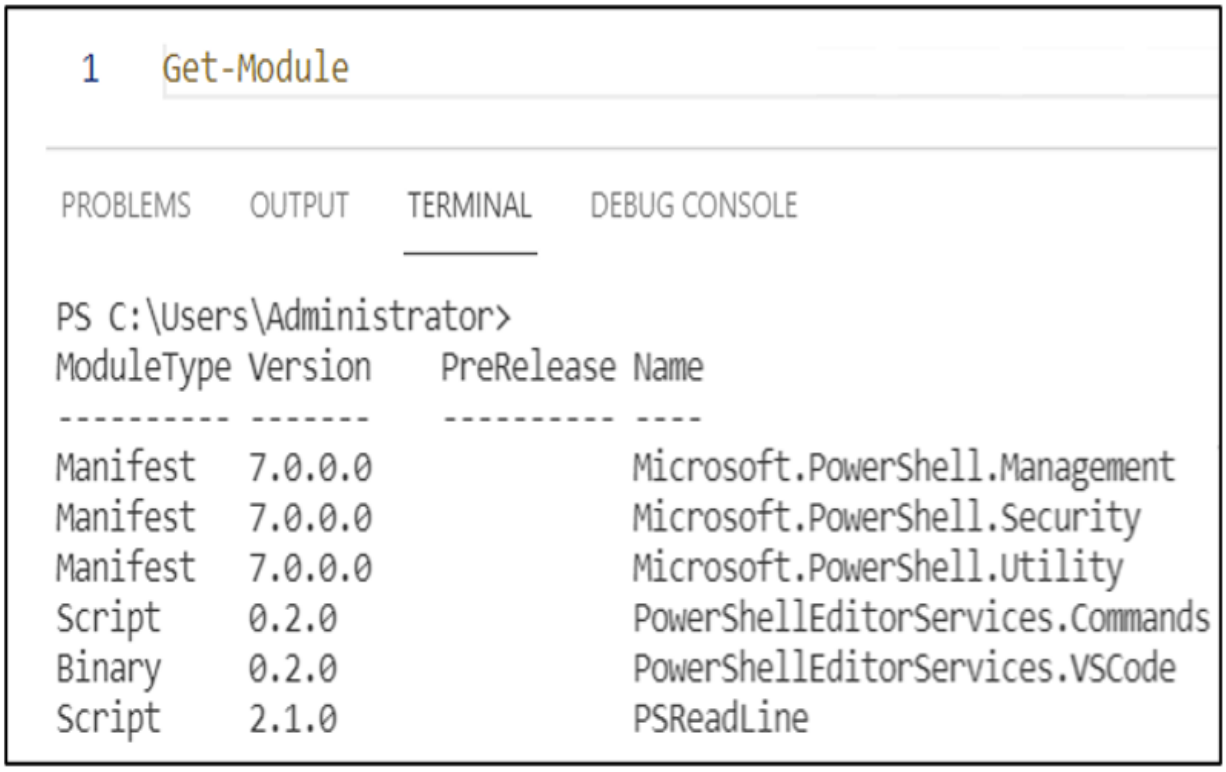
Tee-Object makes a second copy and redirects output. It sends the output in two different directions like a (T) intersection. The data is branched in two different directions. We'll illustrate this concept in the code below.

To start we will use the `Write-Output` cmdlet to write out the text "***IMPORTANT MESSAGE***, this could represent any important message or log able event. We then *pipe* this Output to the `Tee-Object` Cmdlet. The `Tee-Object` will make a branch in the code. One branch will be sent to the output, the other will be captured as a variable called *$second_place*.

We will then write out the value of the *$second_place* variable to validate that the "***IMPORTANT MESSAGE***" was sent to both places.

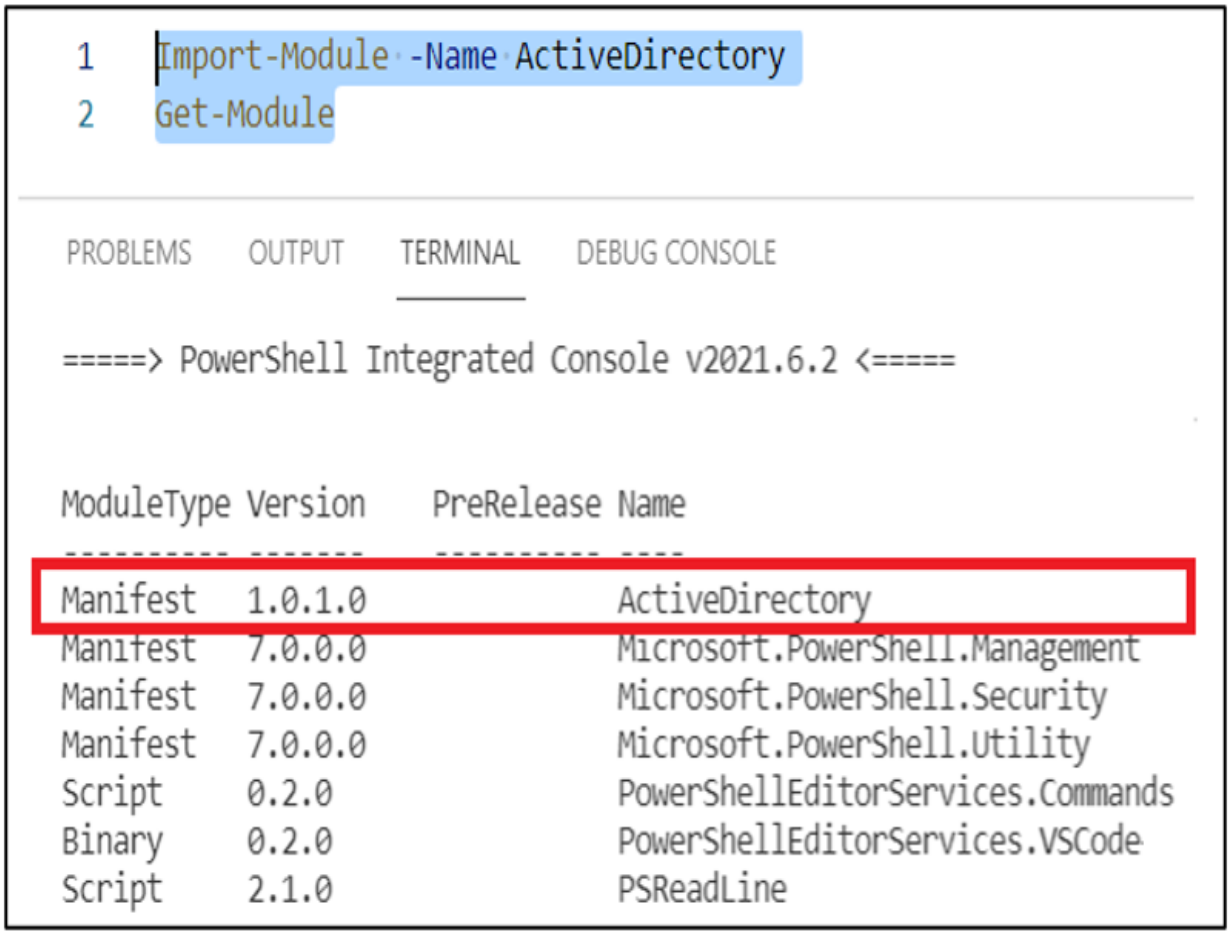
```
Clear-Host
Write-Output "***IMPORTANT MESSAGE***"|Tee-Object -Variable
second_place
Write-Host "The first place it went was to the screen.  The
second place it will go is the 'second_place' variable."
Write-Host "The variable contains: "$second_place
```

Figure 3.11 An example of the Tee-Object cmdlet in action



In addition, we can utilize the Tee-Object cmdlet to send data to places other than a variable. In the Tiny Project code, we utilize the Tee-Object cmdlet to send data simultaneously to the console and to a log file.

Figure 3.12 An example of Tee-Object to send to a log file



One of the parameters available to the `Tee-Object` is the -Append flag. Writing anything to a file, such as the `Tee-Object` we do in this project will overwrite the existing data within the file. Utilizing the -Append flag will append your message to the end of the file instead of overwriting it.

3.4 Get-ADUser

Utilizing Active Directory is a huge part of any System Administrator's job. Microsoft has made this significantly easier with a number of built-in cmdlets expressly for navigating Active Directory.

`Get-ADUser` is a powerful way to pull information directly out of Active Directory. It will, by default, pull the user's: *DistinguishedName, Enabled status, GivenName, Name, ObjectClass, ObjectGUID, SamAccountName,*

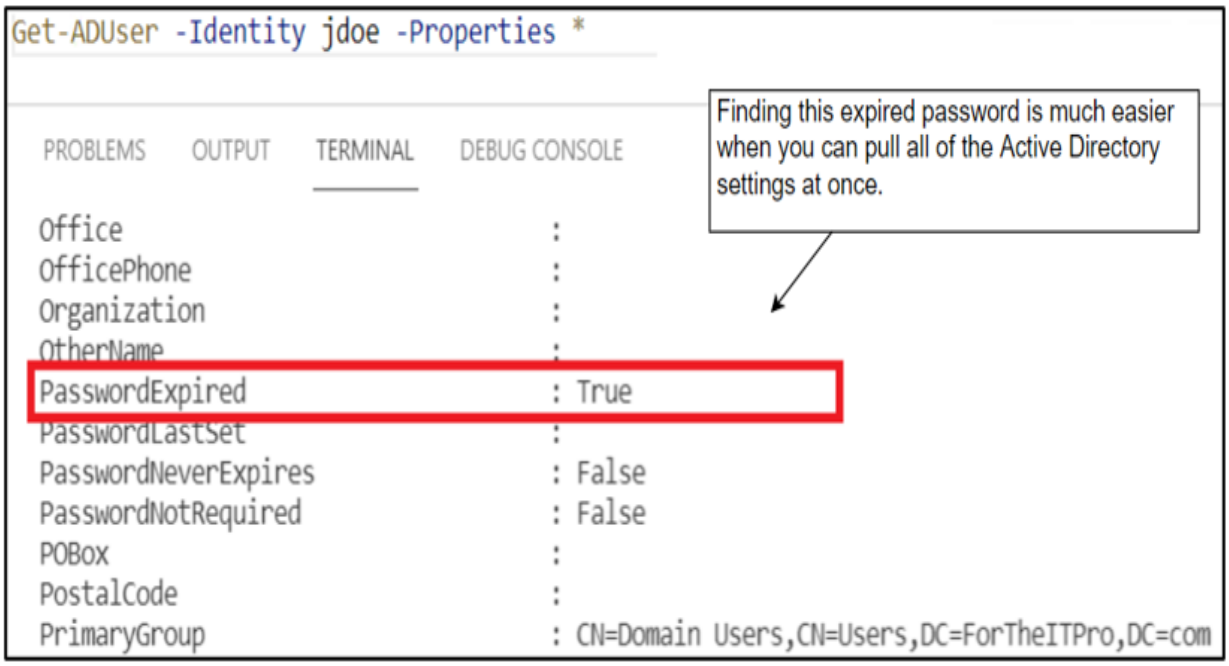
SID, Surname and *UserPrincipalName* from your Active Directory database.

Import-Module

Unless you are running your PowerShell on your Domain Controller it's likely that you may see the error message: The term 'get-aduser' is not recognized as a name of a cmdlet, function, script file or executable…

Because of the sheer bulk and flexibility of the PowerShell language, not all cmdlets are loaded into memory for direct use by default when you install PowerShell 7 (or utilize older built-in versions).

Figure 3.13 Cmdlet not recognized error.



You can resolve this by running the `Import-Module` cmdlet.

```
Import-Module -name ActiveDirectory
```

After the module was imported it's now listed in the `Get-Module` command along with a subset of the cmdlets that are included in the module.

Figure 3.14 List of installed modules with the Get-Module cmdlet

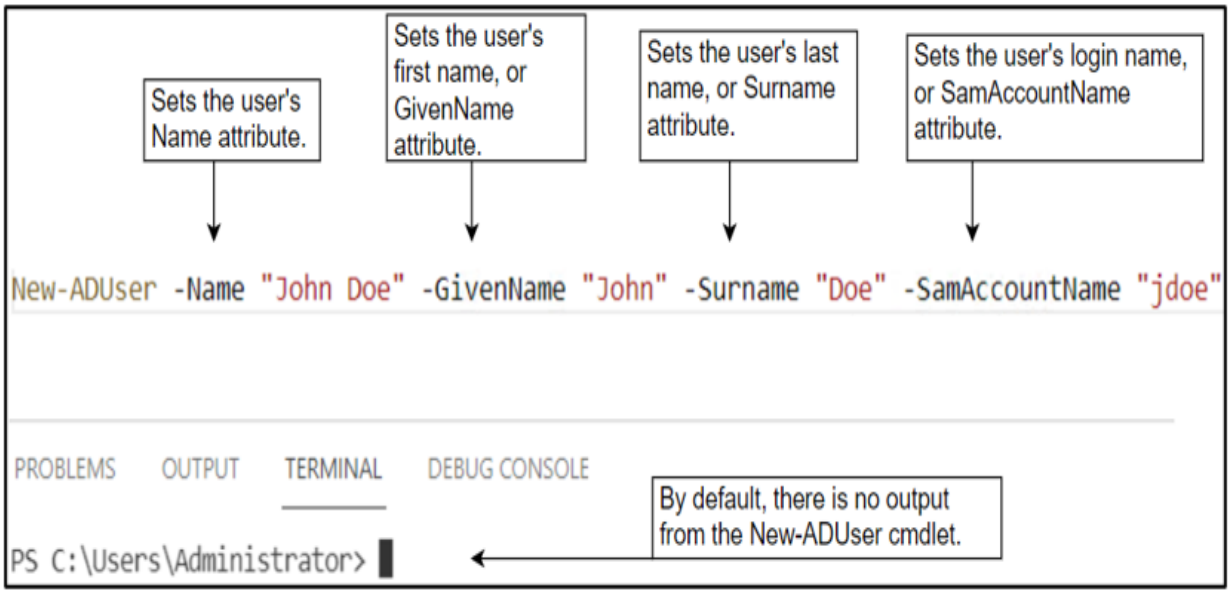
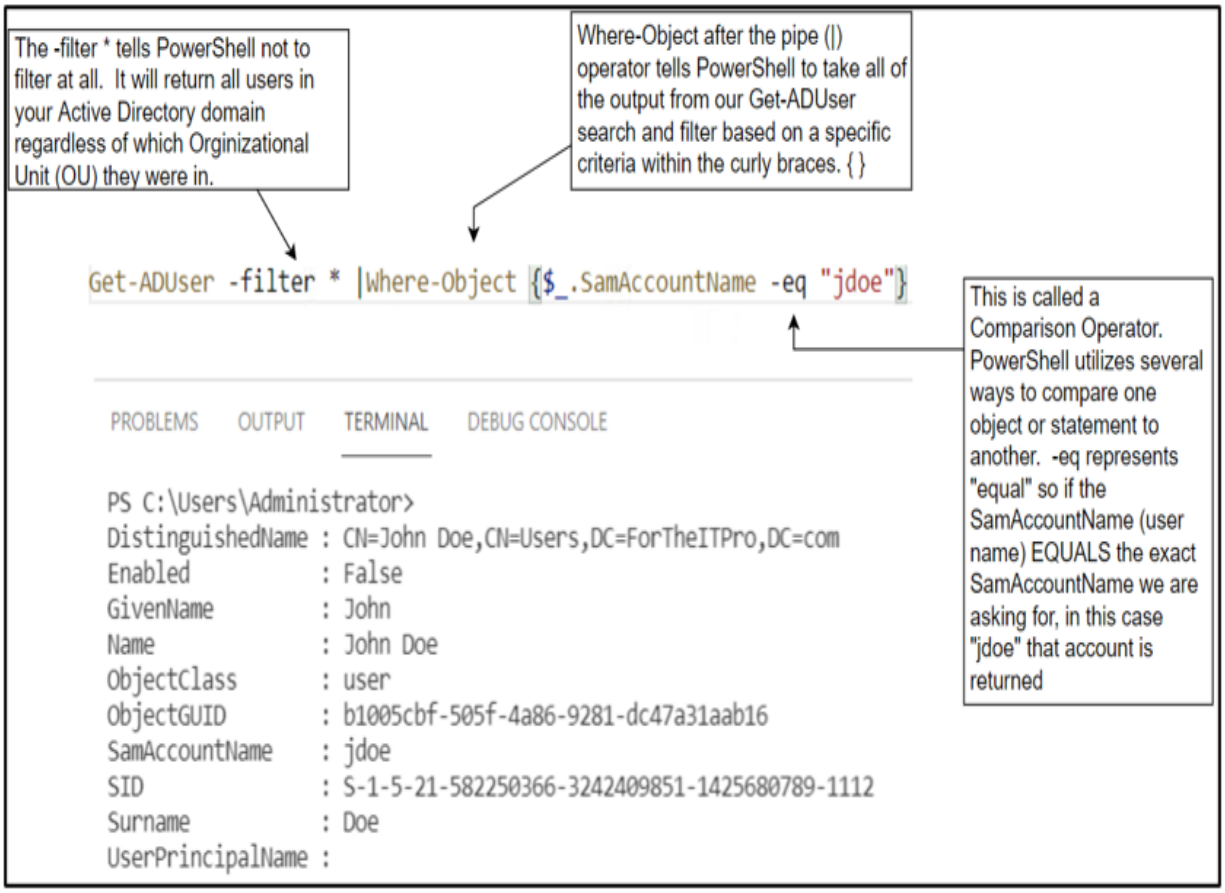


Figure 3.15 List of modules after running Import-Module ActiveDirectory

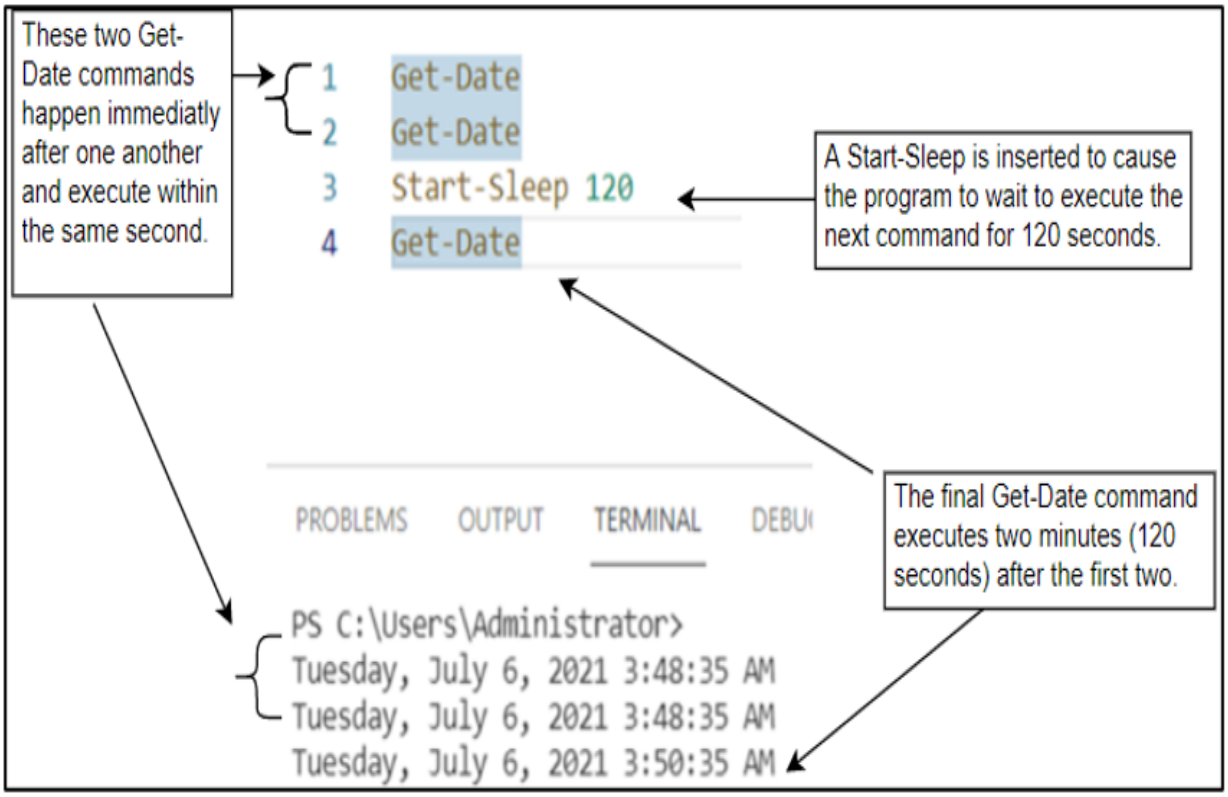


Now the ActiveDirectory Module is imported.

3.4.1 Active Directory Methods

The `Get-ADUser` cmdlet has a number of parameters associated with it. A parameter is a fundamental component of nearly all cmdlets that allows the cmdlet to accept input at runtime. If a cmdlet's behavior needs to change in some way a parameter is usually what allows for this. One of the most powerful of these parameters is the `-Properties` parameter. Active Directory has dozens of unique classifications associated with each Active Directory Object. Several of these, like the *SamAccountName, Name* and *DistinguishedName* are shown by default when you run the command. But you can specify any of the options when you run the command, or you can get them all. Virtually any information you could want to know about a user, computer or group can be found by taking a look at the Active Directory properties.

Figure 3.16 Looking at Get-ADUser properties.



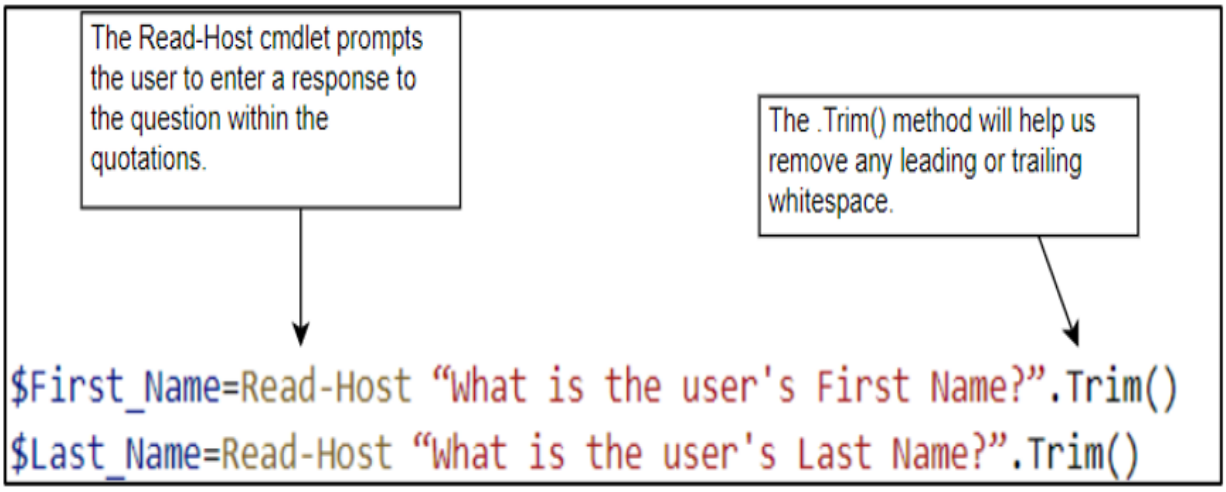
3.5 New-ADUser

`New-ADUser` is a cmdlet in the ActiveDirectory module that allows for PowerShell users to quickly and easily create a new user in your domain.

```
New-ADUser -Name "John Doe" -GivenName "John" -Surname "Doe" -
SamAccountName "jdoe"
```

This creates the user John Doe in your active directory forest.

Figure 3.17 Breakdown of the New-ADUser cmdlet and fields.



3.5.1 Where-Object

There are times where you may be using PowerShell to search an enormous amount of data. For instance, you might be searching your entire domain for a specific user, or a user with a specific value in any given field. For example, you might want to query all of your domain to see which of your users are currently locked out. Depending upon the size of your domain you may be searching hundreds or even tens of thousands of users. What would be really helpful is a way to have PowerShell filter all of those results to only give you the information you care about.

`Where-Object` is exactly the tool you can use to do this.

Note

It is generally considered best practice to "Filter Left" whenever you can. In the example below we are applying the filter to the right of the pipe for demonstration purposes. This will generally be a slower search as more results are returned that PowerShell will need to sort through in with the Where-Object. We will discuss Filter Left techniques, how, when, and why to use them in upcoming chapters.

Figure 3.18 Get-ADUser and Where-Object cmdlets broken down.

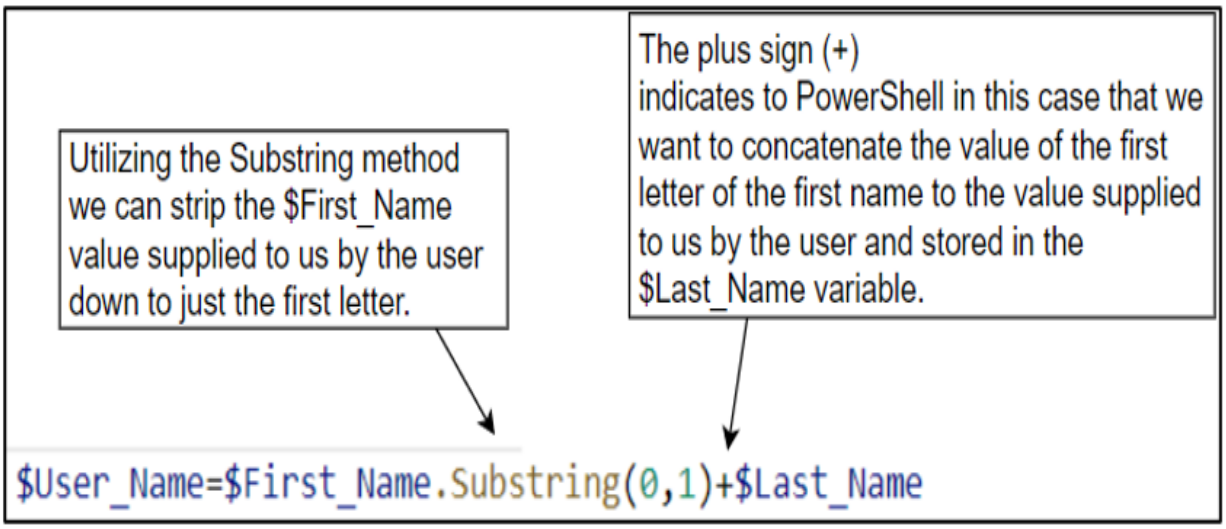


Table 3.1 Comparison Operators

Equal	-eq	The object on the left matches exactly the object on the right
Like	-like	A wild card search that finds part of a string. "*doe", for example, would find any account who's username ended in doe
Greater Than	-gt	Returns a result if the value on the left is greater than the value on the right.
Less Than	-lt	Returns a value if the value on the left is less than the value on the right.

There are many other examples of Comparison Operators that PowerShell can take advantage of, we'll see many more applications of these in future chapters.

$_ What is this strange variable?

The more you utilize PowerShell the more you will see the strange variable defined only as ($_). Simply put the $_ is an automatic variable. This can essentially be used as a placeholder for "Each instance of". In this code:

```
Get-ADUser -filter *|Where-Object {$_.SamAccountName -eq "jdoe"}
```

We asked PowerShell to fetch every Active Directory user in the whole domain. We then wanted to make sure that we only return the single account that matched "jdoe". But PowerShell could be returning hundreds or even tens of thousands of accounts when we have it return every user in Active Directory.

By including the *$_* we are asking PowerShell to compare the *SamAccountName* of each instance and return the one that equals "jdoe".

3.6 Start-Sleep

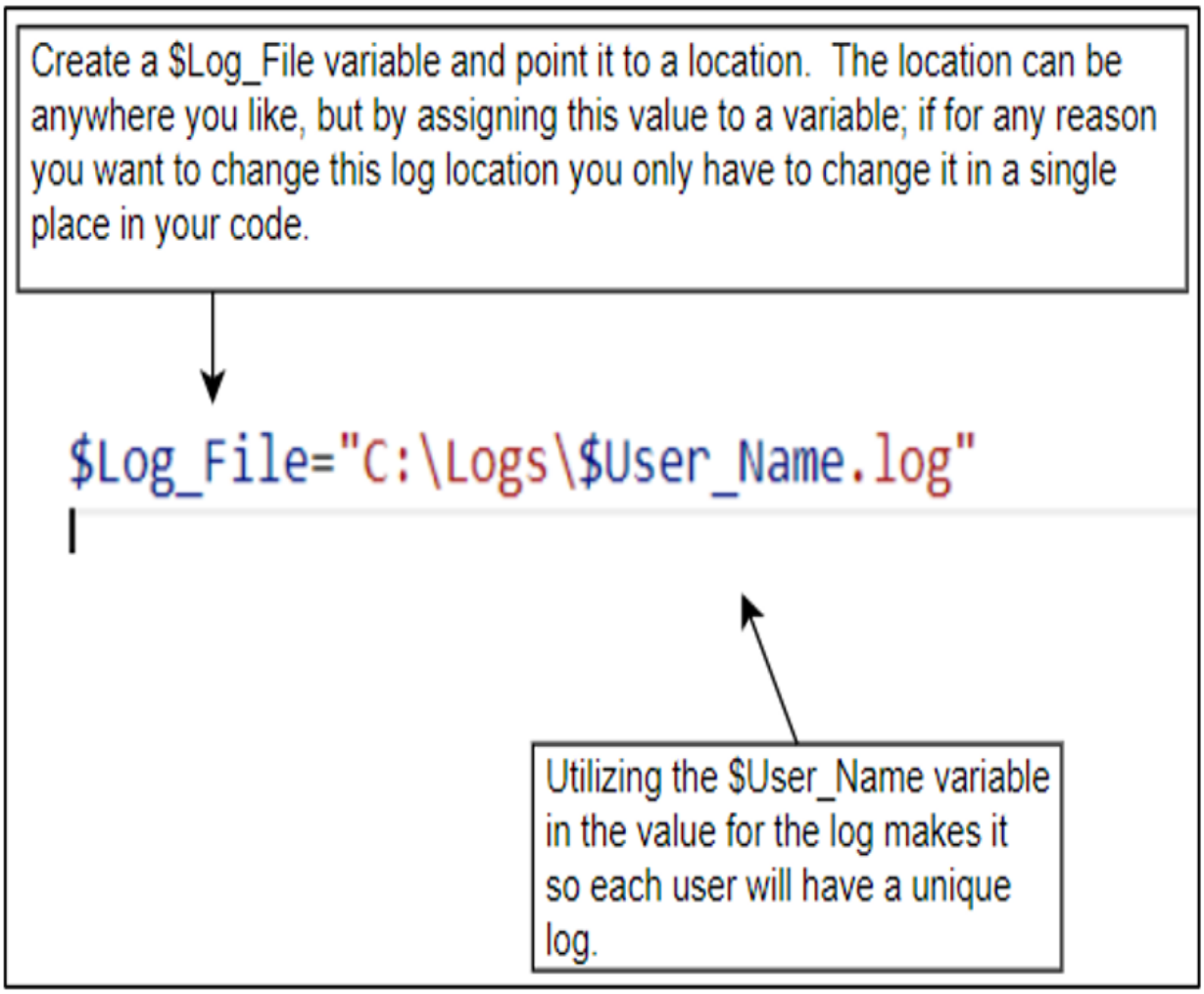
There are times where it might take several moments for an action to occur. If you queried an Active Directory structure like we did above, depending on the size of the domain, it could take several seconds to even a minute or two to pull all of the results. Creating a new user can sometimes take several moments to complete. Then, because that account needs to replicate across all of your domain controllers it may take several seconds for all of your Active Directory clients to be aware of the new user. If we were to create a user then immediately query active directory for that user we might create what's known as a race condition.

A race condition is basically when two things are happening at the same time and you're unsure of which one might finish first. Will the DCs replicate their Active Directories before PowerShell returns a user from its query?

If replication is slower than the query PowerShell might respond that the user doesn't exist. Because when it queried the Domain Controller it was speaking to, the user account had not yet replicated to its database. If the administrator had simply waited for replication to finish, he or she would have received a different answer.

`Start-Sleep` is like a pause button. It puts your script into a timed wait where it doesn't attempt to execute the next command until it has waited the amount of time you requested.

Figure 3.19 Start-Sleep cmdlet in action



3.7 Putting it all together

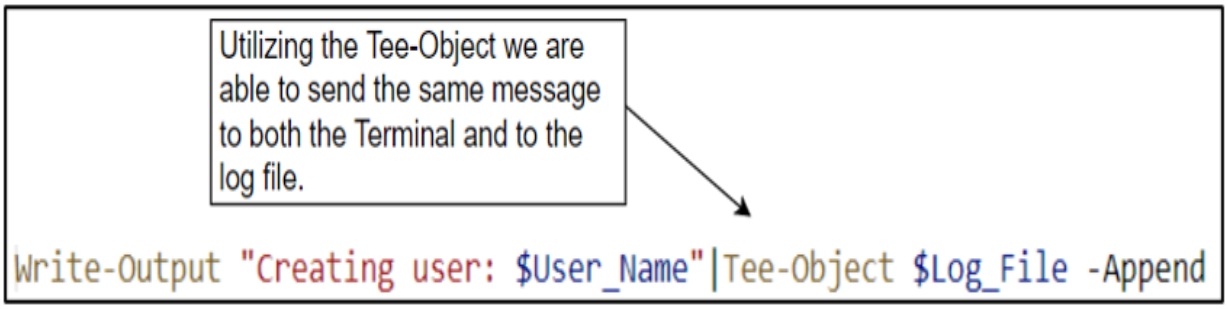
Now that we have learned how each component works in isolation let's see how we are arranging them to build our completed script.

3.7.1 Read-Host

Using the `Read-Host` cmdlet we prompt the operator of our script to enter the First and Last Name of the user they want to create in Active Directory. This allows for more dynamic use of the script because we no longer need to edit the script before we run it modifying the variables beforehand. Now the operator supplies those values each time the script is run.

Because there is a possibility for the operator to hit a space before or after the values he or she enters we also utilize the `Trim()` *method* we learned in the last Tiny PowerShell project to remove any whitespace for each of these variables.

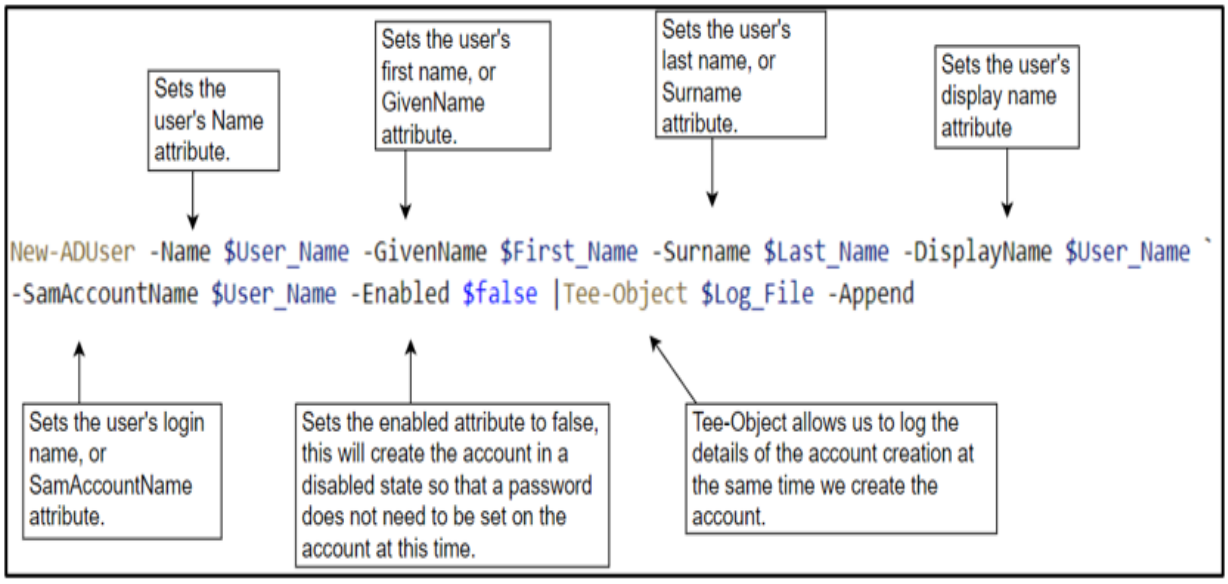
Figure 3.20 Read-Host from Tiny PowerShell Project Code.



3.7.2 Create a Username

With the values of First and Last name supplied to us by the operator we can use the `Substring` *method* and string *concatenation* we learned in the last Tiny PowerShell Project to create a username that takes the first character of the $First_Name and *concatenates* it to the $Last_Name

Figure 3.21 Create Username from Tiny PowerShell Project Code.



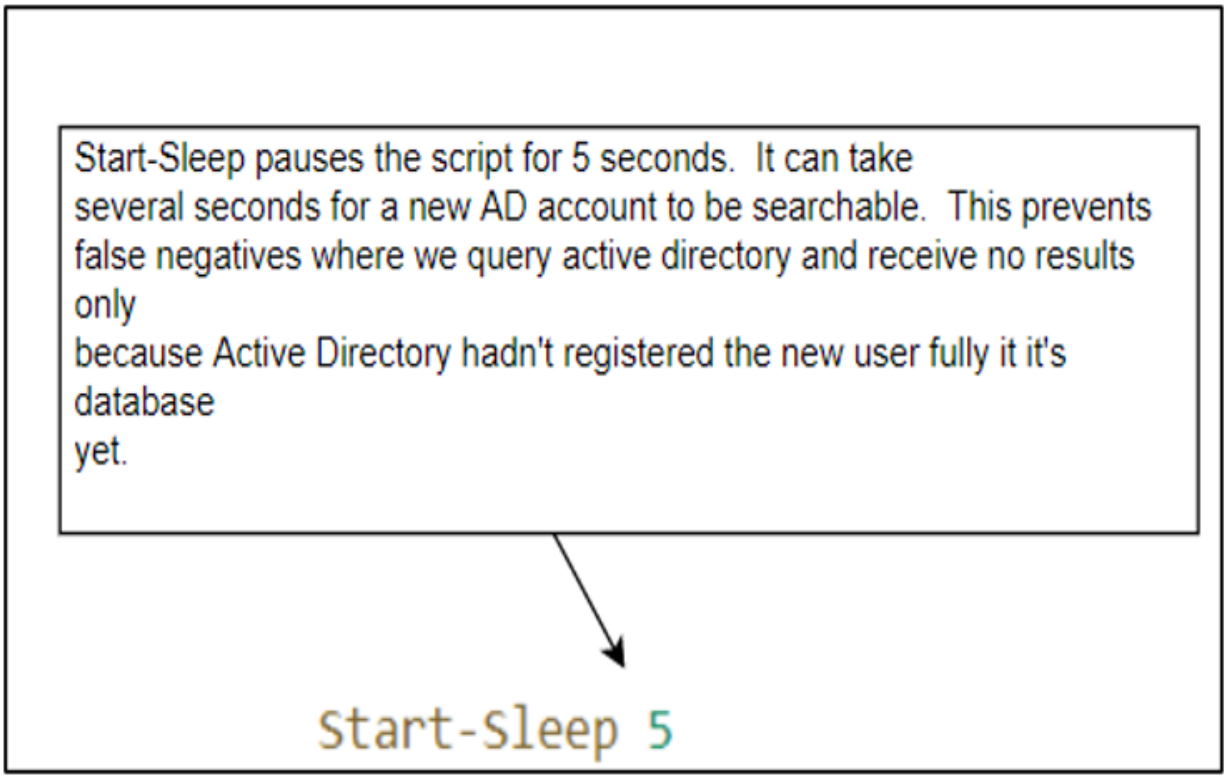
3.7.3 Create a Log File

Logging is very important. It provides a record of who is doing the actions within Active Directory. It also helps when we are trying to troubleshoot an issue. Knowing exactly what the script did can help us find bugs in our script (Our next Tiny PowerShell Project!).

By specifying the path to the Log File as a variable we make it easier to modify this path in the future. If for any reason this path needs to change, we change it in this one spot and all of the logging moves with it seamlessly.

Without the variable, we would need to change it in every instance where we send an output to a log, that may be a lot. Missing even a single instance could cause us to miss critical data in the logs.

Figure 3.22 Using a variable for a log location in our Tiny PowerShell Project code.



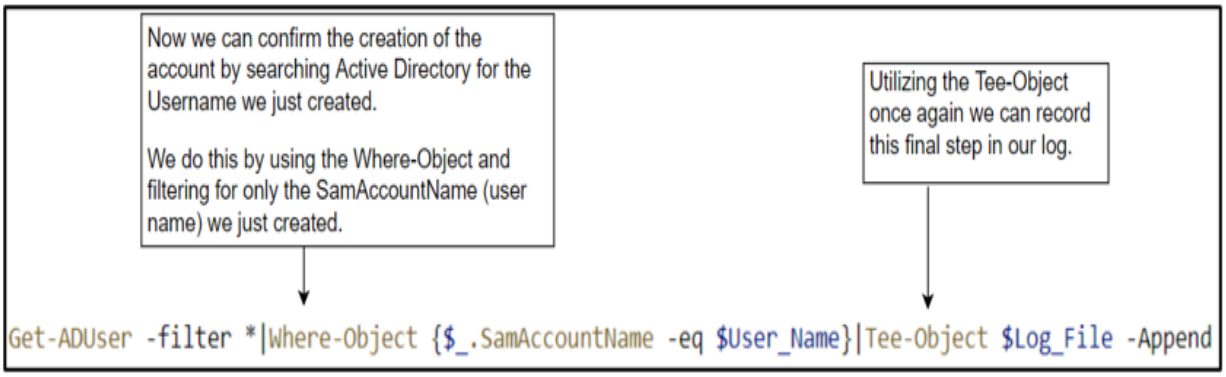
3.7.4 Write-Output

So far there is not a lot of data being passed to the operator. A long period of silence can cause a Windows Admin to begin to wonder if the process is working or if the process is hung.

Therefore, it's important to try to give the operator some indication the process is not hung. We can do this by writing our steps to the console. In addition, we want to make sure we capture the commands that are being run in the log. Because we don't want to duplicate our effort, we can utilize the `Write-Output` and the *pipe* mechanic combined with the `Tee-Object` cmdlet to simultaneously send output to the log and output for the operator.

We send our first message to the Operator and the Log confirming that we are creating a User for the $User_Name we created earlier.

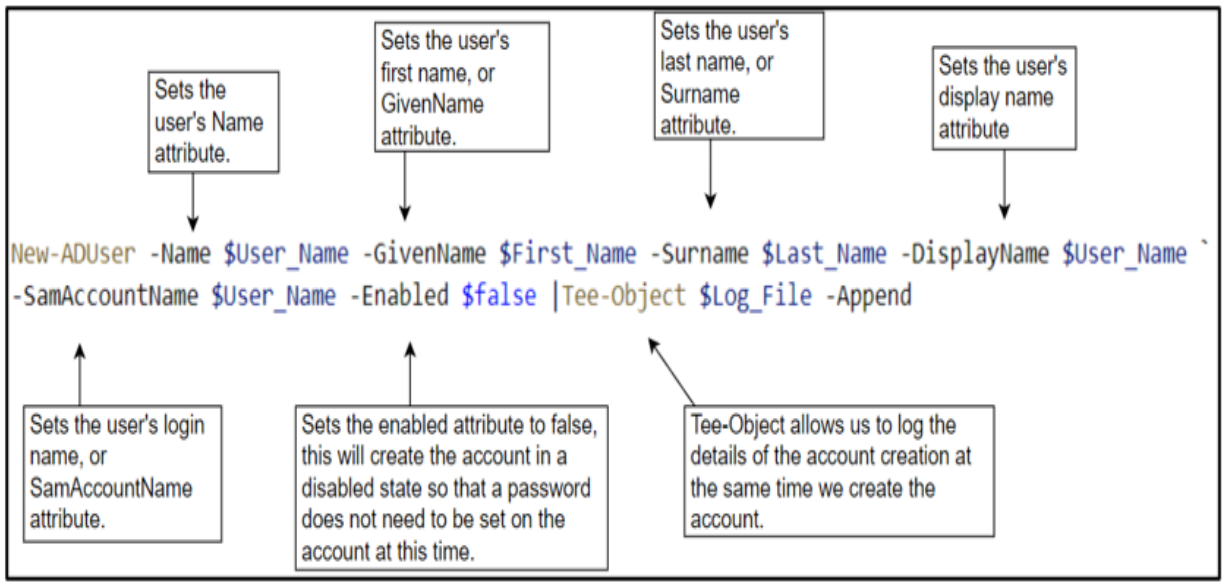
Figure 3.23 Using the Write-Output and Tee-Object cmdlets in our Tiny PowerShell Project code.



3.7.5 New-ADUser

Utilizing the Active Directory cmdlets we can create new Active Directory users with a single line of Code. We create a new user in Active Directory taking the information supplied to us by the Operator of the script. We're able to propagate the *Given Name* (first name), *Surname* (last name), *SamAccountName* (User Name), *Display Name* (User Name) and with the *pipe* mechanic and `Tee-Object` cmdlet we can send the output of the New-ADUser cmdlet to our screen and to the log file.

Figure 3.24 New-ADUser from Tiny PowerShell Project code



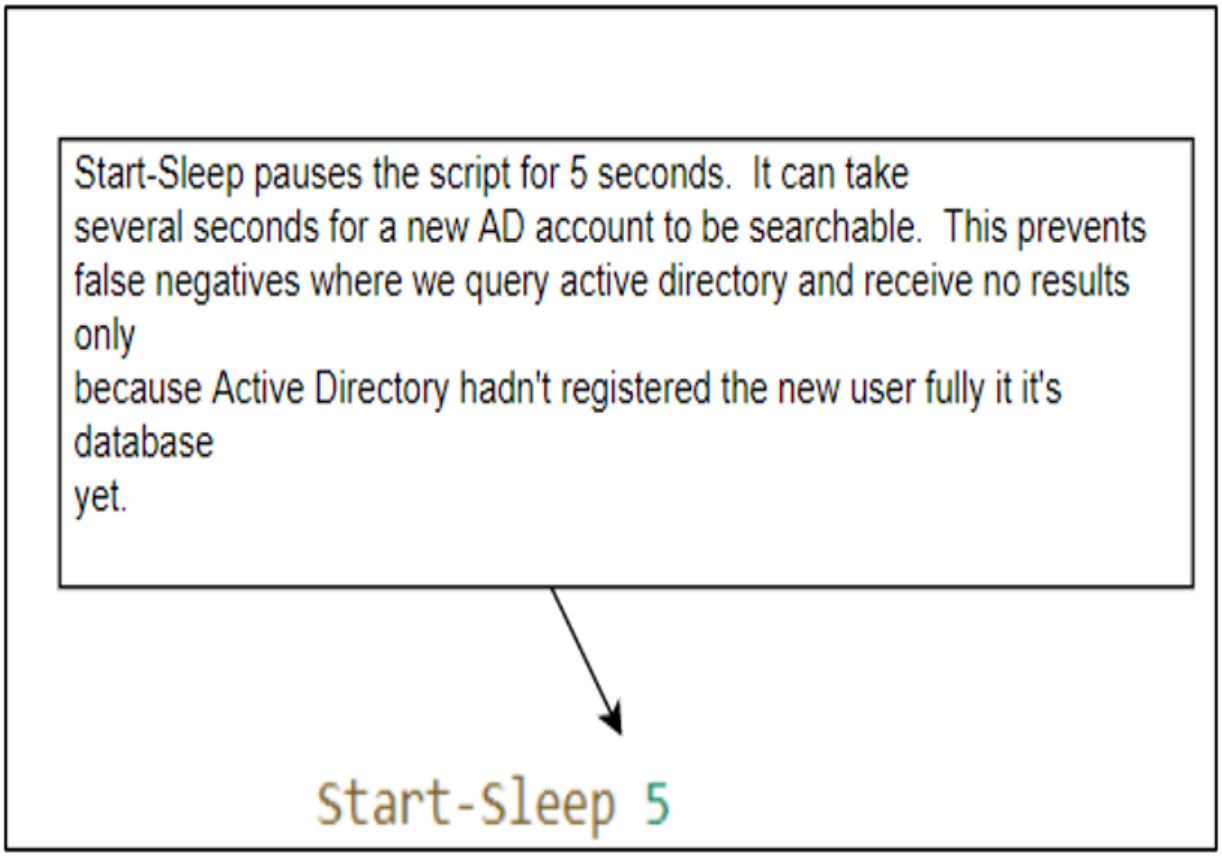
3.7.6 Start-Sleep

We'll want to confirm the process worked. The easiest way to do this is to look in Active Directory to validate that AD now has a user that matches the Username we just supplied it. But because, it's not quite instant; we utilize the `Start-Sleep` cmdlet. PowerShell will process our lines of code sequentially without hesitation. Because of this, it would execute the `New-ADUser` cmdlet. Then, as soon as Active Directory completed the creation of the user it would immediately attempt to `Get-ADUser`.

Because of Replication across AD databases, it's possible we might be querying the database before the entry was found creating the appearance that the command did not work. We can resolve this situation by simply allowing some time for Active Directory replication to catch up.

The `Start-Sleep` cmdlet simply allows PowerShell to execute the command and then wait for the specified amount of time before instantly moving to the next command. To help us eliminate the race condition.

Figure 3.25 Start-Sleep from Tiny PowerShell Project code.



3.7.7 Get-ADUser

The very best way to confirm that the new user was created in Active Directory is to check Active Directory for our expected user. We can query all of Active Directory with the `-filter *`. But this could give us hundreds, or thousands of users returned. We don't want to have to sort through that haystack looking for our new needle.

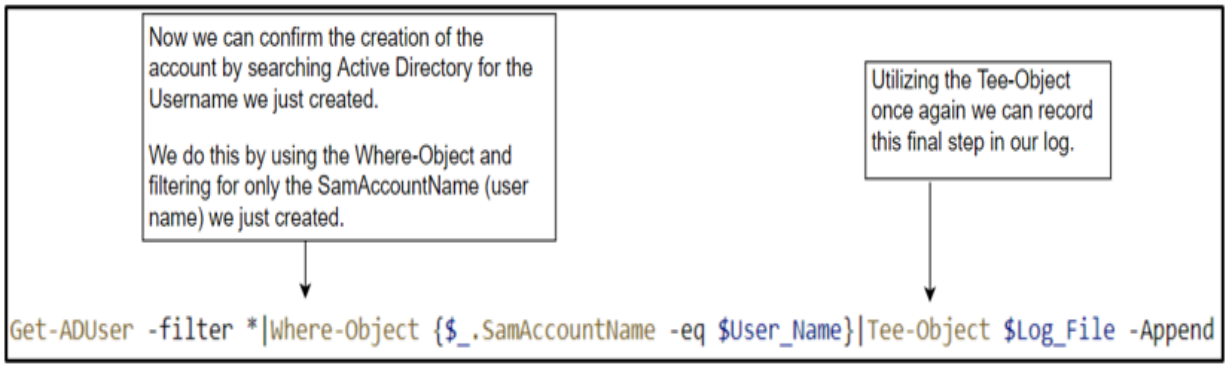
Instead, we can have PowerShell do the work for us. By utilizing the *pipe* mechanic and the `Where-Object` cmdlet we can have PowerShell pull all of the users in Active Directory and then provide us only the User Object that matches the username we supplied it. Utilizing another *pipe* mechanic and a Tee-Object cmdlet we can send this information simultaneously to the operator and the log to confirm both at time of creation and for anyone later reviewing the log that the user was successfully created.

Creating user accounts are one of the most common tasks that an IT Professional are likely to get. Each user on the system will need an account to use it, there's just no way around that. Today we have begun to automate this process. Imagine how much more time you'd have to work on more challenging issues if you didn't look over at the stack of new hires needing accounts and know that it was going to eat up a couple of hours out of your day?

The value of PowerShell should be evident. In upcoming chapters this value will unfold in so many news ways. In later chapters we'll be building upon the skills you learned here to not only create user accounts, but manage those user's groups, and email addresses with no more information than you used in this script.

In addition to the speed in which you created these users, creating new users with a script also makes you far more consistent. There are lots of moving pieces in creating a user account, and while none of them are particularly hard, it does leave a lot of room for simple mistakes. Forgetting to put in a field, typing a username incorrectly, typos in general. All of these common mistakes can simply be eliminated from your environment by consistently using scripted solutions.

Figure 3.26 Get-ADUser and Tee-Object from Tiny PowerShell Project code.



3.8 Summary

- We can store user supplied information in variables using Read-Host
- With combinations of Read-Host and Write-Host we can make menus that allow for more granular control of our scripts.
- The PowerShell pipe mechinic takes the output of a command or series of commands and inputs that into a nother command or series of commands without having to store the intermediate values as variables.
- Write-Host has no output, but Write-Output does.
- Tee-Object can be used to easilly duplicate and send messages to multiple sources.
- Active Directory has its own set of PowerShell cmdlets to help us manage our domain. We can get access to all of these by using the Import-Module command.
- Start-Sleep can be used to help us elimnate race conditions

4 Bugs! (troubleshooting common script issues)

This chapter covers:

- Introduction to debugging
- Navigating the debugging tool provided by PowerShell and VS Code
- Identifying and correcting syntax errors using messages and markings
- Finding and correcting logical errors using the Break and Write-Host Cmdlets

Now that you've got a few scripts in your bag of tricks it's important for you to learn a new skill, debugging. The fact is, writing the code is the easy part. Debugging is the hard part, figuring out why you're getting result A instead of B.

Even if you have no real desire to write original scripts, this chapter is important! Because there are any number of environmental differences on one domain to the next. The scripts provided in the Tiny PowerShell Project book are designed to run out of the box in most environments. But you can still run into issues implementing them and this chapter will give you the tools you need to empower you to get around your implementation problems and see the value from these Tiny PowerShell projects in your own environment.

4.1 Making Sense of the Red

Unlike a lot of programs one might run on a Windows platform that give you vague error messages that leave little to work with such as "An Error has occurred."; PowerShell gives useful examples of what the problem is and how to fix it.

Errors in PowerShell show up red and it can be intimidating to look at a screen full of red and not know what it's trying to tell you. Let's look at a

common error you might see if you're trying to use a saved script on a hardened PC (Figure 4.1).

Figure 4.1 Close look at an error message

The execution policy mentioned in the error message in Figure 4.1 is a safety feature that details the conditions under which PowerShell can execute a script on your system. By implementing an execution policy, you can prevent all scripts or unsigned scripts from being run on your PC, thus helping to prevent the execution of malicious scripts.

In our case, however, we know that this is not a malicious script so we can remove this restriction. We utilize the Set-ExecutionPolicy cmdlet to change the execution policy on our system.

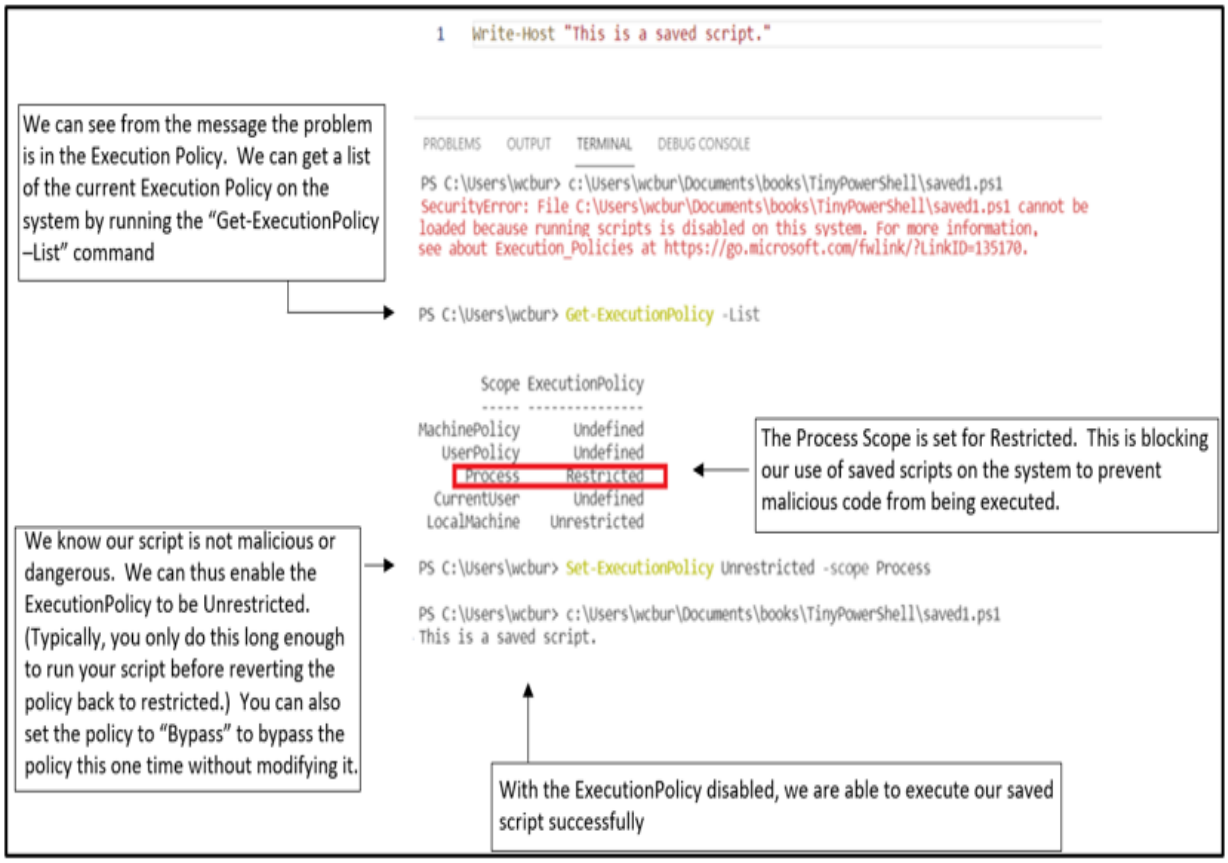
```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

This will unrestrict your computer from running unsigned scripts. You can re-enable this safety feature by running the Set-ExecutionPolicy cmdlet again.

```
Set-ExecutionPolicy -ExecutionPolicy Restricted
```

Running the above command will re-enable the Execution Policy restriction on your PC.

Figure 4.2 Using the information in an error message to find and resolve the issue



Error messages like those explored in figures 4.1 and 4.2 help us correct errors and learn more about PowerShell as we develop our scripts. Next, let's look at how we can further de-bug scripts in PowerShell utilizing VSCode and PowerShell error messages to get from broken to working.

4.2 Project code

Warning:

Unlike all of the other code in this book, this code is intentionally left broken. On the whole, it should look very much like some of the other scripts we have been working with this far and through the rest of the book. But when you run it, as written, it will have several syntax errors and even a logical error.

Don't worry if you can't understand what the code is doing. We'll get to loops and if statements later in the book. For now, we're just trying to debug the code so it runs.

If you're familiar with the song 100 bottles of beer on the wall, you should get a kick out of this script (Figure 4.1). When we've finished debugging the code and finally get it to run it will print out the full lyrics to the screen for the whole song; Of course it won't do it just yet.

The "project" in this Tiny PowerShell Project is not about what the script will do for us; but rather how to make you, the reader, a more capable scripter and able to debug common issues when you see them.

Figure 4.3 Original Tiny PowerShell Project Code

```
$i="100"
do{
    $object="bottles"
    $objec2="bottles"
        If($i == 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
    $song="`t$i $object of beer on the wall.`n"
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    $i2=$i-1
    Write-Host $song
    $i--
} until ($i -lt 1)
```

When we run this code VS Code does its best to help us identify the syntax issues we are having to help us to get the code to run.

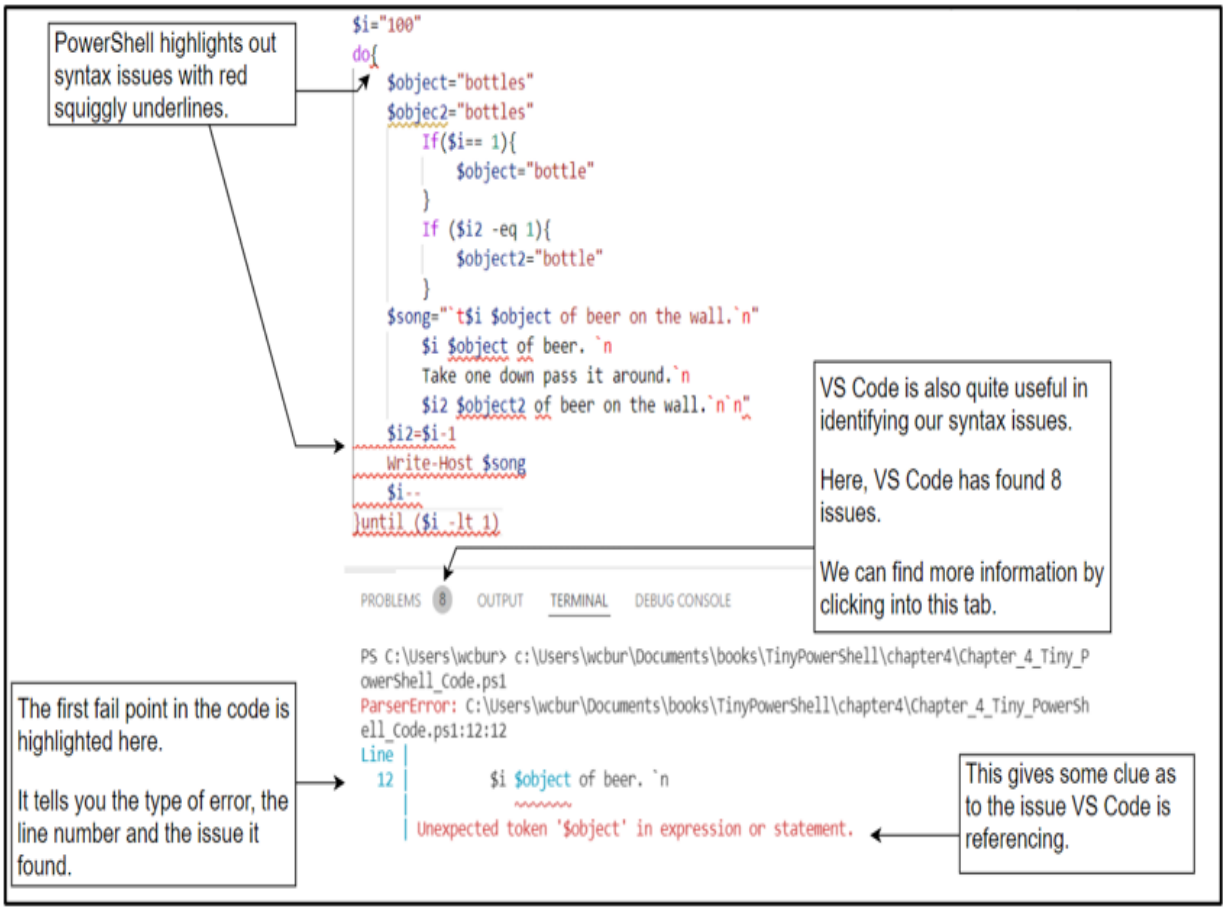
Debugging in PowerShell 5.x vs PowerShell 7.x

PowerShell 5.1 could be run in VS Code; however, it also had a built in Windows PowerShell ISE built on .Net Framework. The troubleshooting techniques will be similar to what we use in PowerShell 7.x, but VS Code makes this significantly easier. Since, unlike version 5.1, PowerShell 7.x does not run on PowerShell ISE, all bug detection and remediation for this book will be done with VS Code.

4.2.1 Tips and Tricks

When it comes to debugging code there's really two types of errors, syntax issues and logical issues. A syntax issue is where the interpreter doesn't understand our code and can't run it. These types of errors used to be a nightmare, and a programmer could spend days looking for that one colon that should have been a semicolon. However, with today's modern scripting compilers and interpreters most syntax issues are generally pretty easy to find (Figure 4.4).

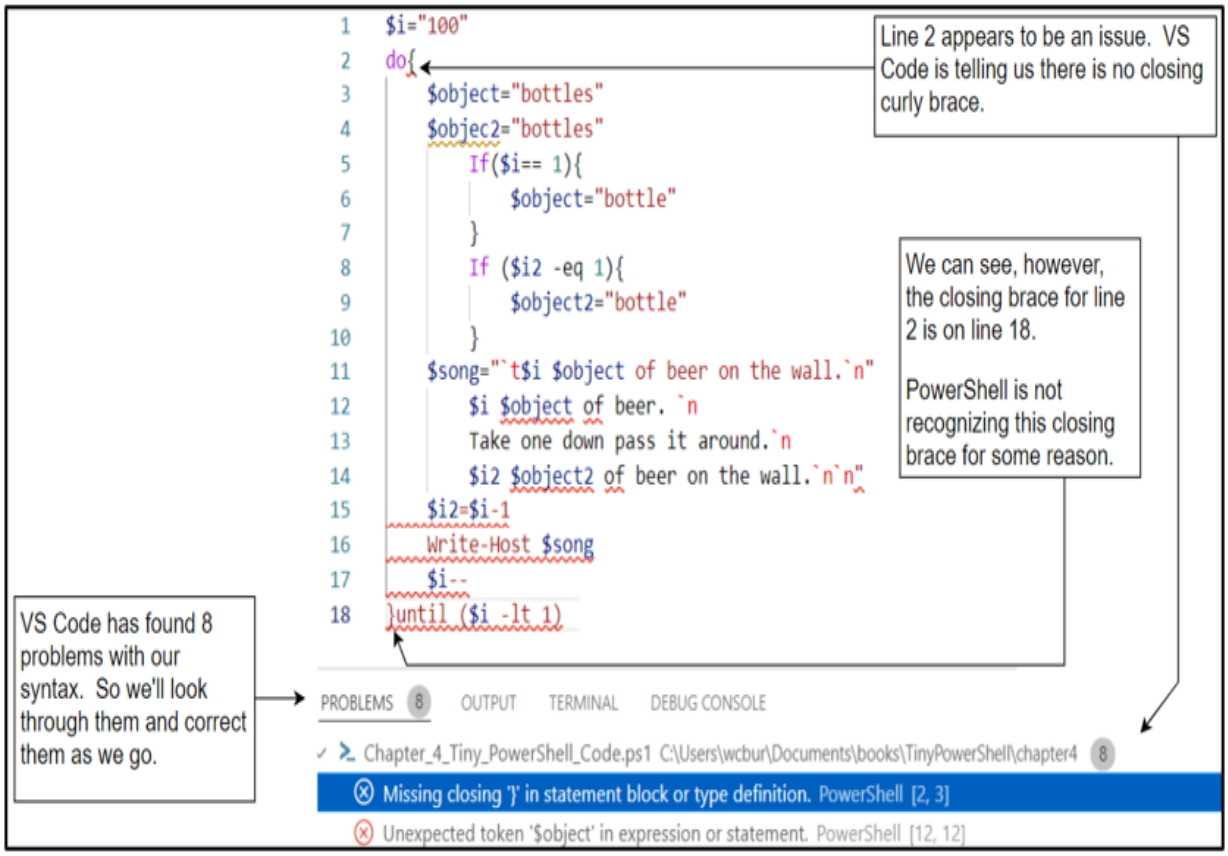
Figure 4.4 Finding Syntax issues with VS Code



4.3 Syntax issues

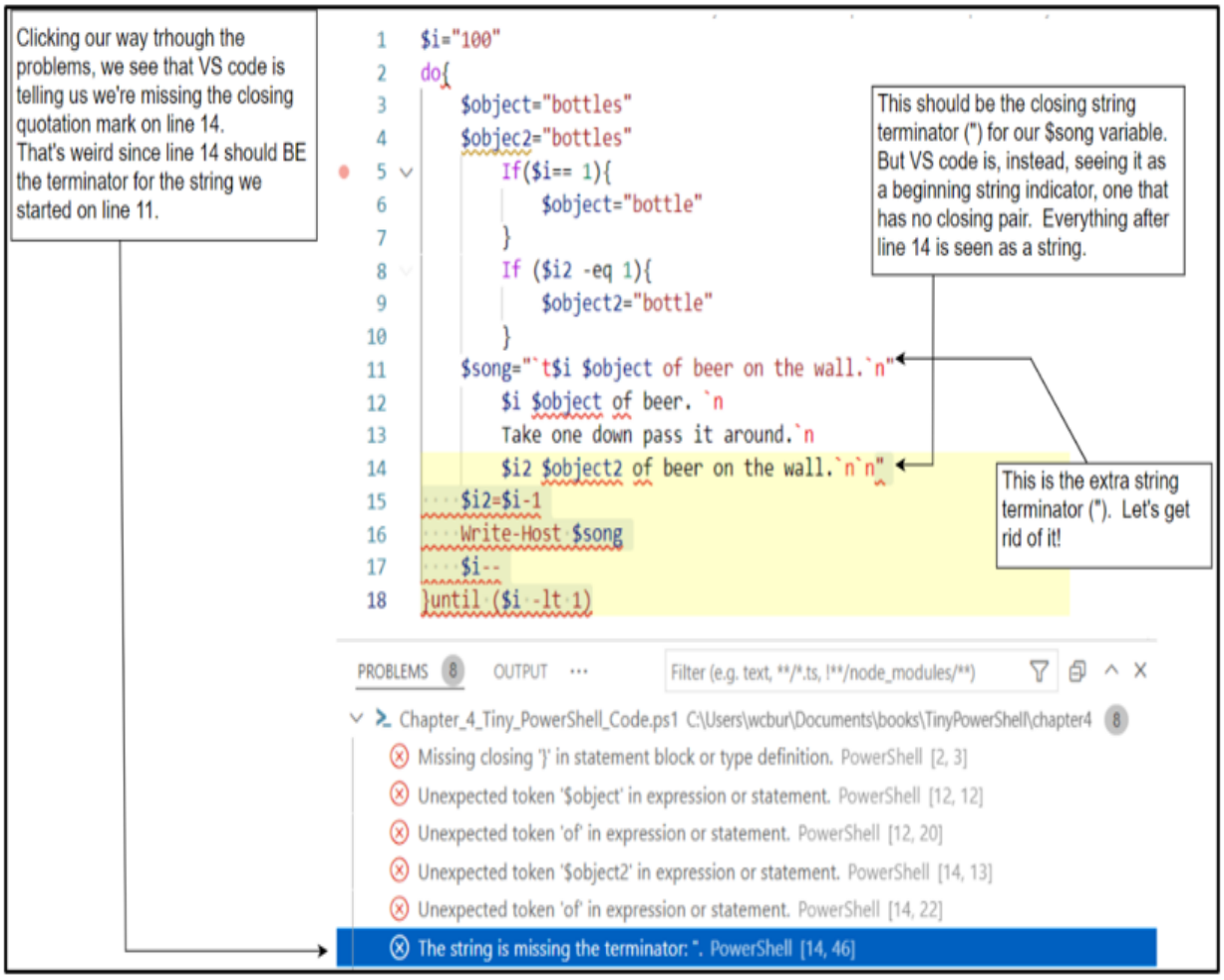
VS Code has identified issues the syntax of our code. A good place to start your debugging is by clicking on the "Problems tab" and quickly finding the problem areas (Figure 4.5).

Figure 4.5 Examples of Syntax issues that don't seem to make sense.



Looking at the output of the "Problems" tab we see that PowerShell is informing us there is a missing curly brace somewhere in our code. But there's obviously an issue with this code as we clearly have the closing brace included. For some reason VS Code is not seeing it correctly. We can step our way through the issues detected by VS Code to see if anything jumps out at us (Figure 4.6).

Figure 4.6 Resolving common Syntax issues with VS Code.



Here is our first catch. VS Code is telling us we have a missing closing terminator (“) at line 14. But the terminator at the end of line 14 *should* be the closing terminator for the string we started at line 11, not an opening one.

Looking at between line 11 and line 14 we can see the culprit. There is an extra quotation mark at the end of line 11. VS Code saw the $song variable as:

```
$song="`t$i $object of beer on the wall.`n"
```

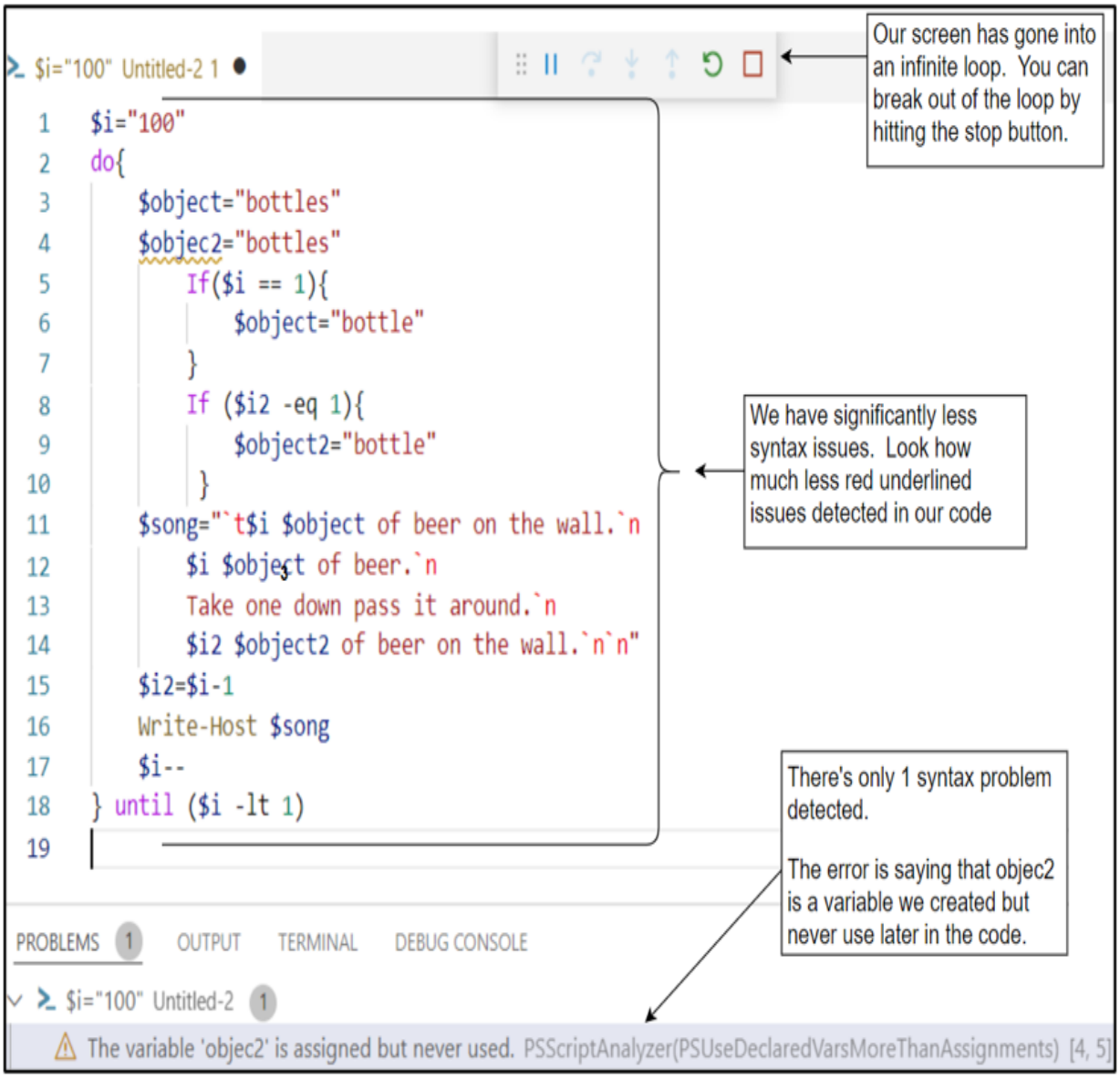
Line 12-14 is no longer seen as part of the $song variable. The quote at the end of line 14 is seen as the *start* of a new sting. So, everything beyond this point is seen as a string. That’s the reason that it didn’t see the closing curly brace on line 18.

We can correct the code so that it looks like this:

```
$i="100"
do{
    $object="bottles"
    $objec2="bottles"
        If($i == 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    $i2=$i-1
    Write-Host $song
    $i--
} until ($i -lt 1)
```

When we run this version of the code there are significantly less syntax issues (Figure 4.7). But now we're stuck in an infinite loop! We can stop the execution by hitting the red square (stop) button on VS Code.

Figure 4.7 Resolving final Syntax issue.



The remaining syntax issue tells us that objec2 is assigned but never used. We are using $object2 several times within the code. But it appears when we created the variable in line 4 we forgot the "t" in object and accidently created the variable as objec2 instead of object2. This is a simple fix.

```
$i="100"
    do{
        $object="bottles"
        $object2="bottles"
            If($i == 1){
                $object="bottle"
            }
```

```
            If ($i2 -eq 1){
                $object2="bottle"
             }
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    $i2=$i-1
    Write-Host $song
    $i--
} until ($i -lt 1)
```

When we fix the code, we no longer see any VS Code identified issues, nor do we see any underlined red in the window. However, the code still will not run successfully. When we run it we see two error messages:

- One about the term = not being recognized.
- One about the -– operator only working on numbers.

What are these errors if they're not errors in syntax? These are logical errors.

4.4 Logical issues

Now the remaining issues in the code are logical issues. These are a little tricker to find. Fortunately, we have some tools that will help us diagnose the issues step by step.

4.4.1 VSCode BreakPoints

We can utilize VSCode and the debugging window to try our code step by step. This will help us check that we have variables set as we expect them to be as we step through the code. Hopefully, we can identify our logical mistakes and figure them out.

Add Breakpoint

A breakpoint is a way we can tell PowerShell to stop executing our code when it gets to a specific point in our script. It's a good way to be able to

step through our code as we can run a section then break out of it making sure that each step does what we expect it to. If we insert a break into our code, we will temporarily take care of that infinite loop that is forcing us to manually stop our code.

Let's also call each of our variables to make sure that they hold the value we expect them to.

Before our break we have the variables:

$i

$object

$object2

$i2

Let's check to see what PowerShell has for these variables. First, we will add Write-Host statements to check the values in our variables. Often when a script fails it is because the value within the variables is not what we think it should be. Validating the input and output of these variables can be a critical first step to debugging.

In addition to this we will also utilize VS Code to insert a breakpoint where execution of the script will pause until we manually continue it. This allows us to check our script step by step. Figure 4.8 and 4.9 detail adding the breakpoint.

```
$i="100"
do{
    $object="bottles"
    $object2="bottles"
        If($i == 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
        }
        Write-Host "variable i is: $i"
        Write-Host "variable object is: $object"
```

```
    Write-Host "variable object2 is: $object2"
    Write-Host "variable i2 is: $i2"
       $song="`t$i $object of beer on the wall.`n
         $i $object of beer.`n
         Take one down pass it around.`n
         $i2 $object2 of beer on the wall.`n`n"
     $i2=$i-1
     Write-Host $song
     $i--
} until ($i -lt 1)
```

Figure 4.8 Using VSCode debugger to create a breakpoint.

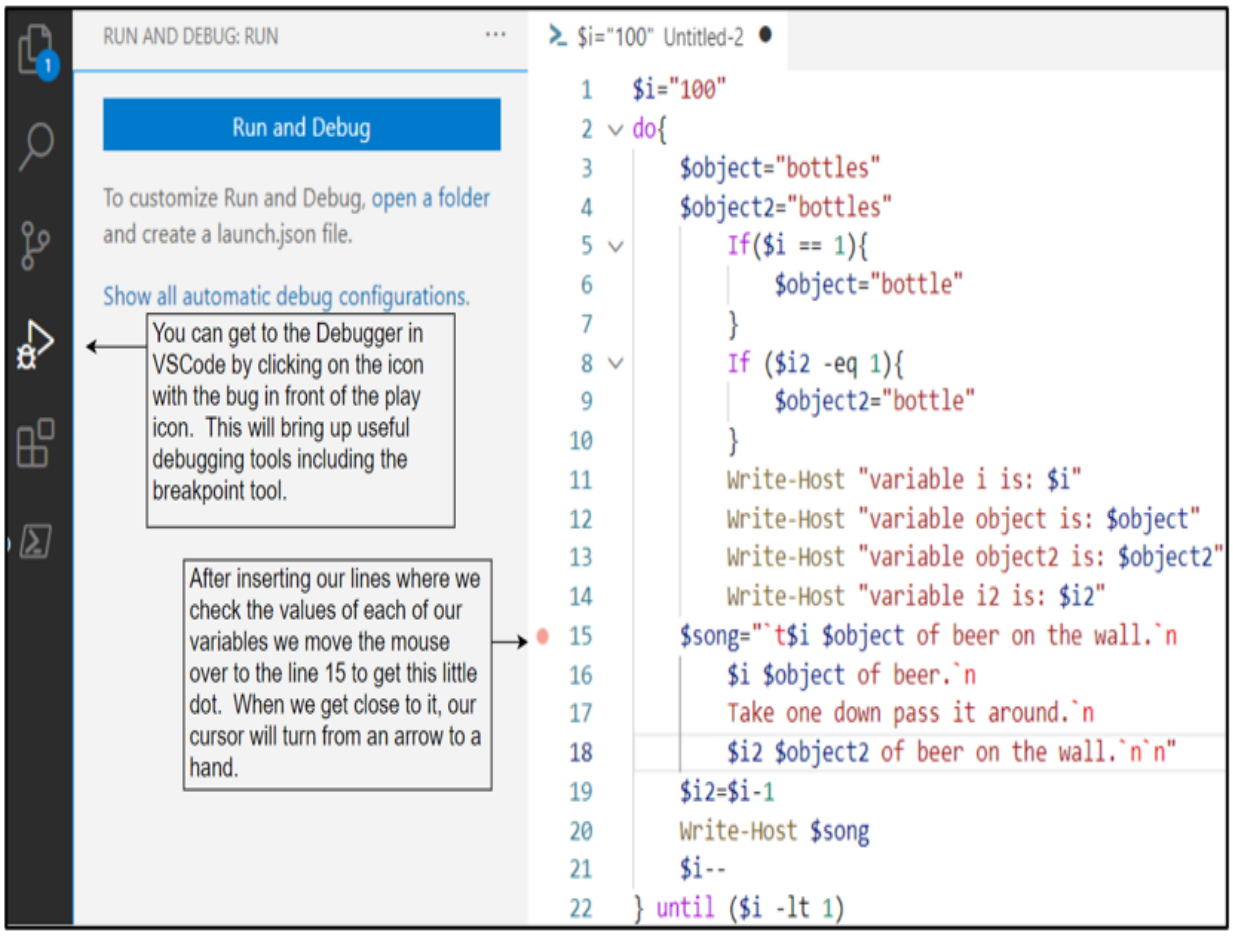
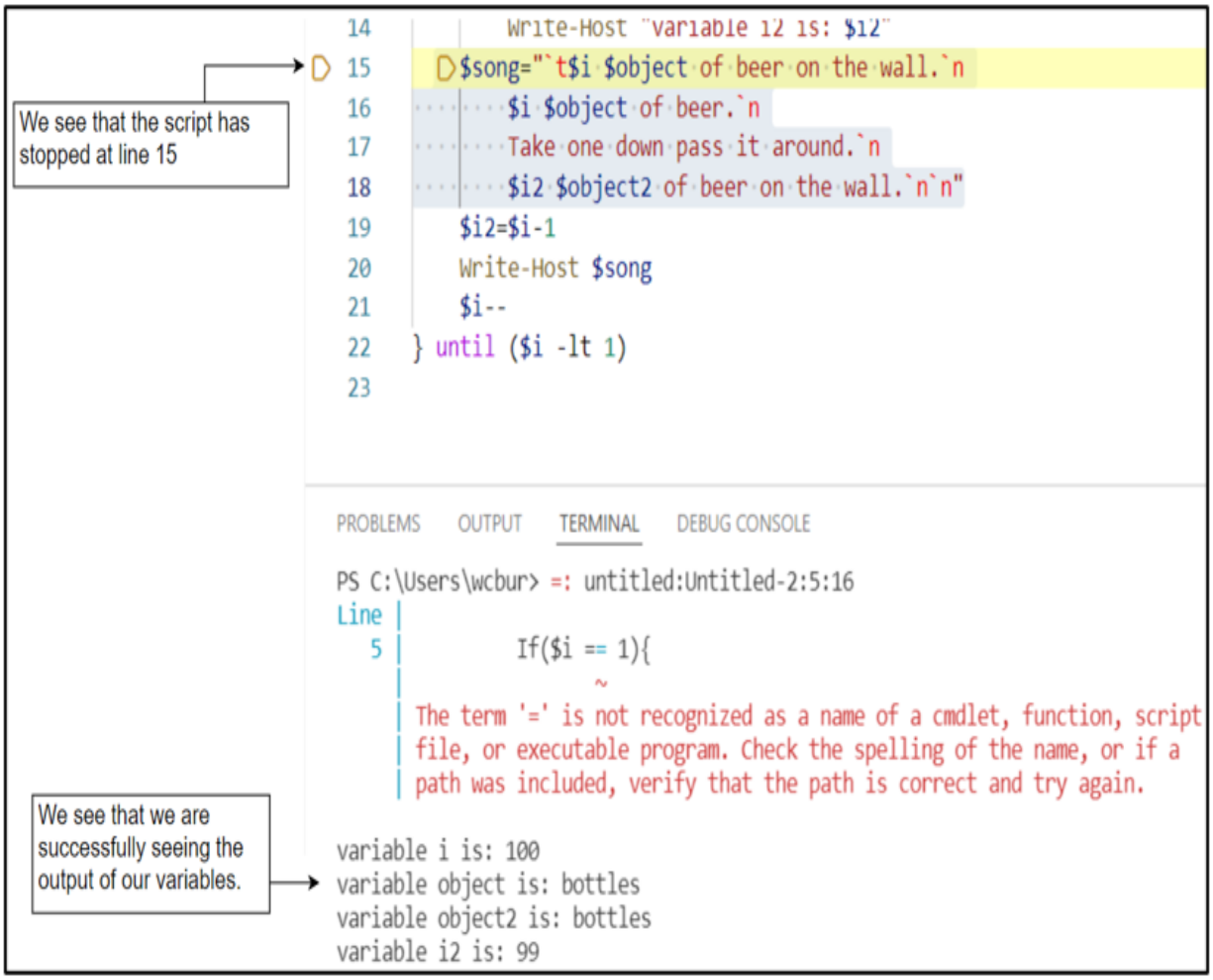


Figure 4.9 Adding a VSCode Breakpoint

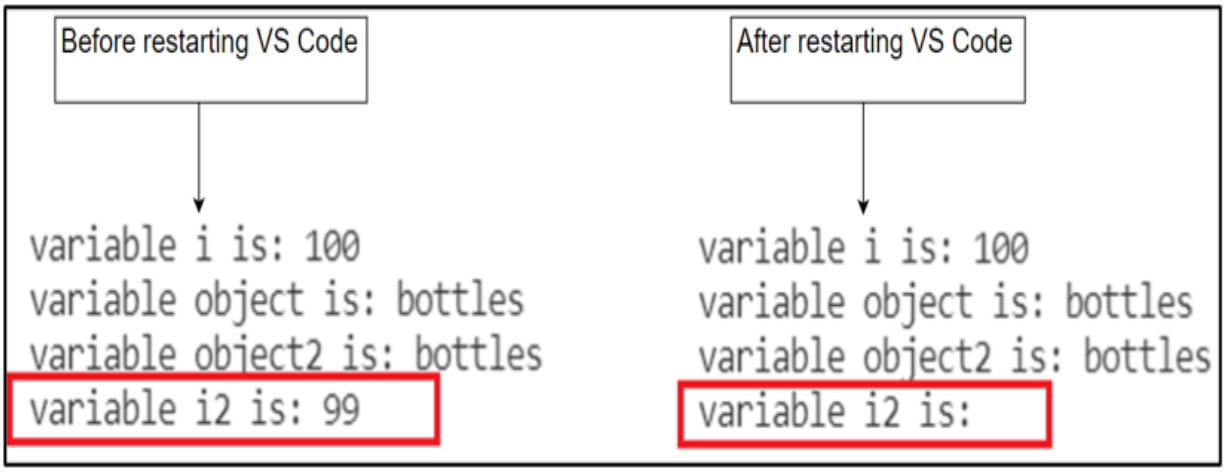
Figure 4.10 Using VS Code Breakpoints



phantom variables

Depending on when you run this code you can see a very strange behavior. Sometimes when you run this code you might see a value 99 returned for $i2. Other times when you run it you might see no value for variable $i2 (Figure 4.11).

Figure 4.11 Example of non-cleared variable in session.



This is caused because, when you assign a variable, it's stored in the session it was first called in. If you had executed the code before we put the `breakpoint` in, PowerShell would have gone far enough through the code to assign a value to `$i2`. So, when we ask PowerShell what the value of `$i2` is, it looks to see if the variable is defined and if it is, it returns the value.

However, if we restart VS Code (and start a new PowerShell session), the break stops the code before PowerShell can assign a value to `$i2` and it has no value assigned so it returns nothing or `NULL`.

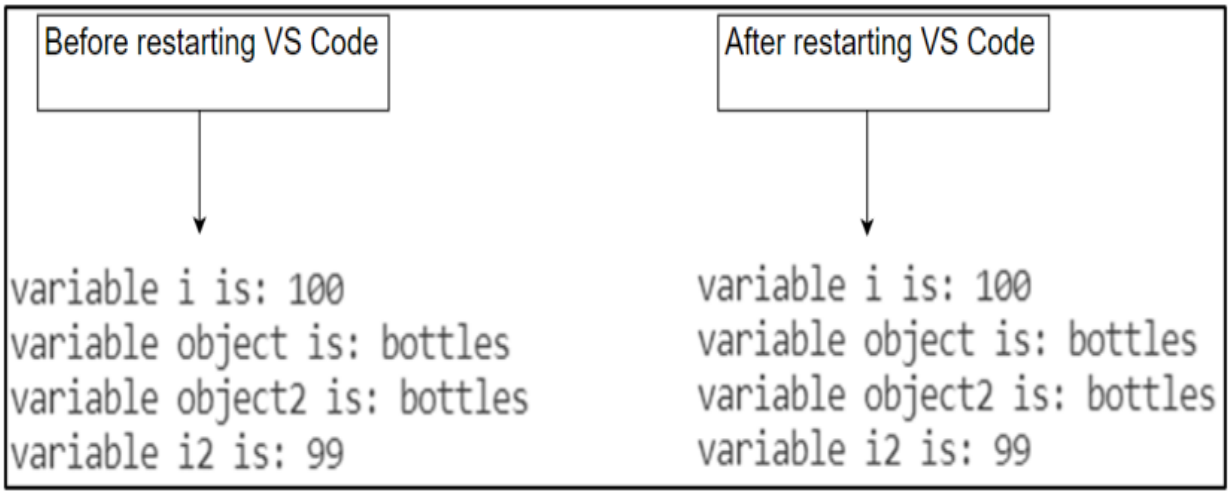
When you're debugging it's a good idea to always clear your variable values, so you don't have old values you're not expecting show up.

We can fix the issue we're seeing with the null value for `$i2`, by simply moving line: `"$i2=$i-1"` somewhere above our break. So, let's move it to just above the section we first use $i2.

```
$i="100"
do{
    $object="bottles"
    $object2="bottles"
    $i2=$i-1

        If($i == 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
```

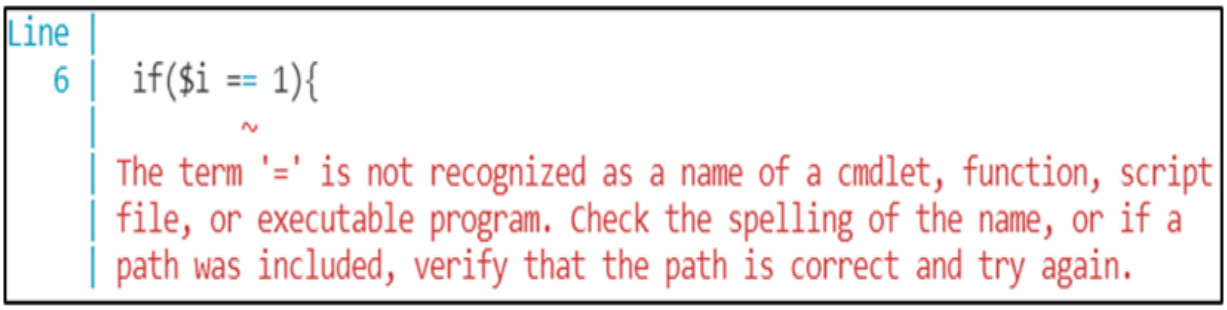
```
        Write-Host "variable i is: $i"
        Write-Host "variable object is: $object"
   Write-Host "variable object2 is: $object2"
   Write-Host "variable i2 is: $i2"
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    Write-Host $song
    $i--
} until ($i -lt 1)
```

Figure 4.12 Resolving phantom variable issue.



Those starting values look great now (Figure 4.12). But we're still seeing a couple of errors in the output. First look at this one (Figure 4.13):

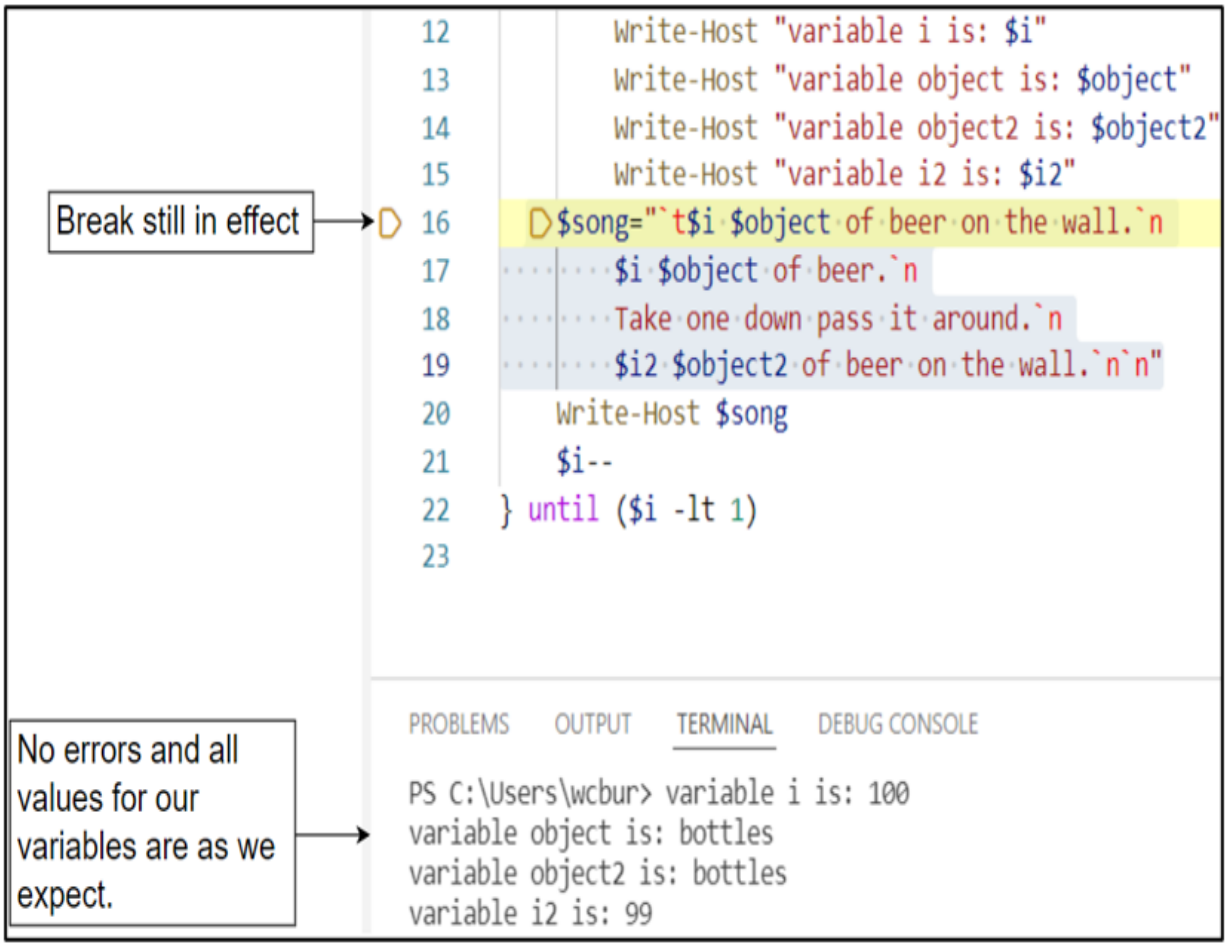
Figure 4.13 Issue with PowerShell Comparison Operators.



The error is telling us on line 6 the term "=" is not recognized. This is in our if statement where we're checking to see if the value of $i is equal to 1. But if you remember from chapter 3, PowerShell, unlike some languages like C, uses -eq to denote if a value is equal not the "==" we used here. Let's fix that!

```
$i="100"
do{
    $object="bottles"
    $object2="bottles"
    $i2=$i-1

        If($i -eq 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
        Write-Host "variable i is: $i"
        Write-Host "variable object is: $object"
   Write-Host "variable object2 is: $object2"
   Write-Host "variable i2 is: $i2"
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    Write-Host $song
    $i--
} until ($i -lt 1)
```

Figure 4.14 All Issues Resolved?



VS Code sees no problems and the starting variables all look like what we'd expect (Figure 4.14).

```
$i="100"
do{
    $object="bottles"
    $object2="bottles"
    $i2=$i-1

        If($i -eq 1){
            $object="bottle"
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
         Write-Host "variable i is: $i"
          Write-Host "variable object is: $object"
    Write-Host "variable object2 is: $object2"
    Write-Host "variable i2 is: $i2"
```

```
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    Write-Host $song
    $i--
} until ($i -lt 1)
```

Stepping through Breakpoints

But what happens when we step through the breakpoints? When we utilize VSCode's debugger, we can set breakpoints manually. VSCode also inserts automatic breakpoints to help us identify issues with our code. We can use the Step tool to step through those breakpoints to see if our code executes (Figure 4.15).

Figure 4.15 Stepping through breakpoints

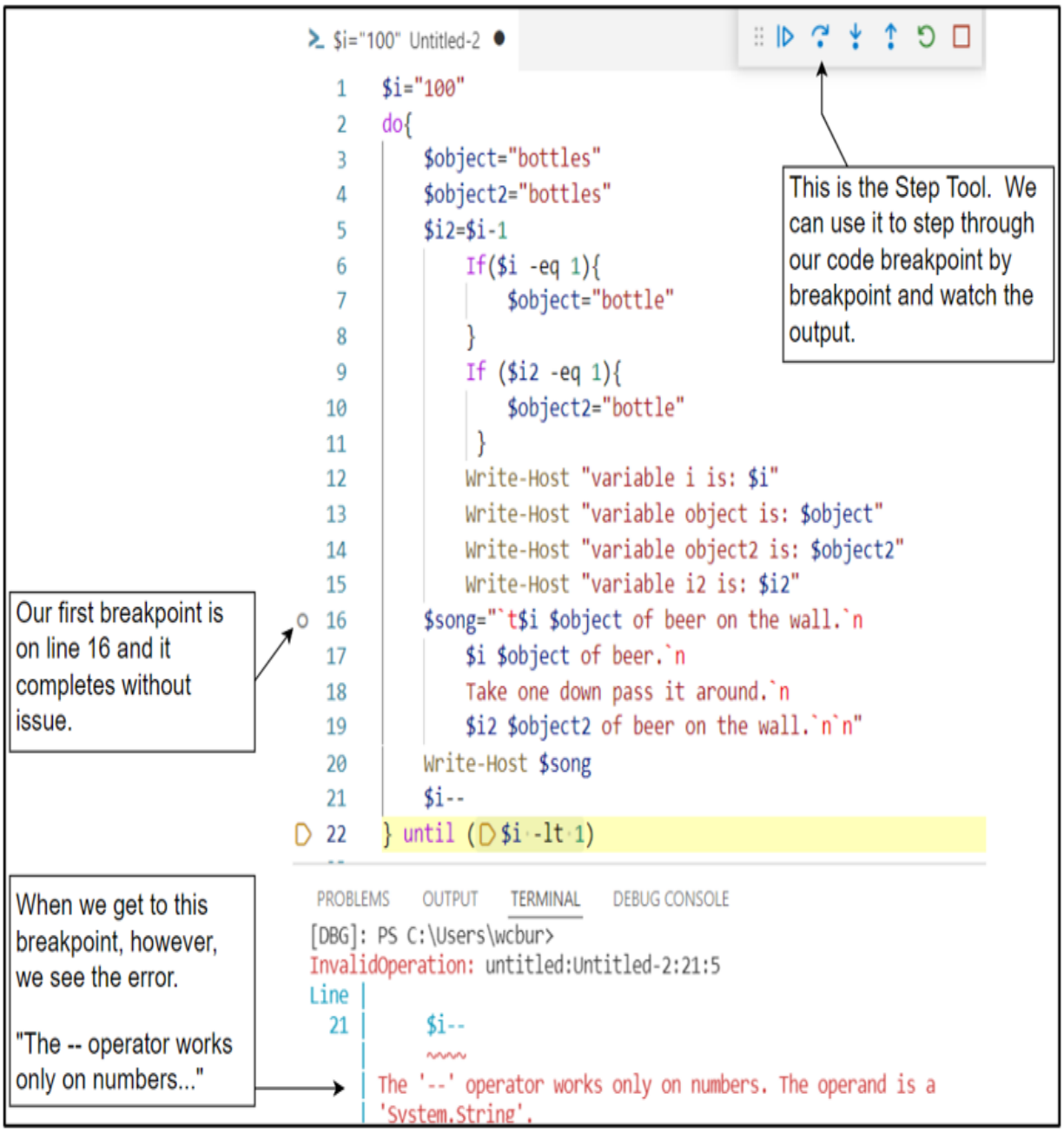
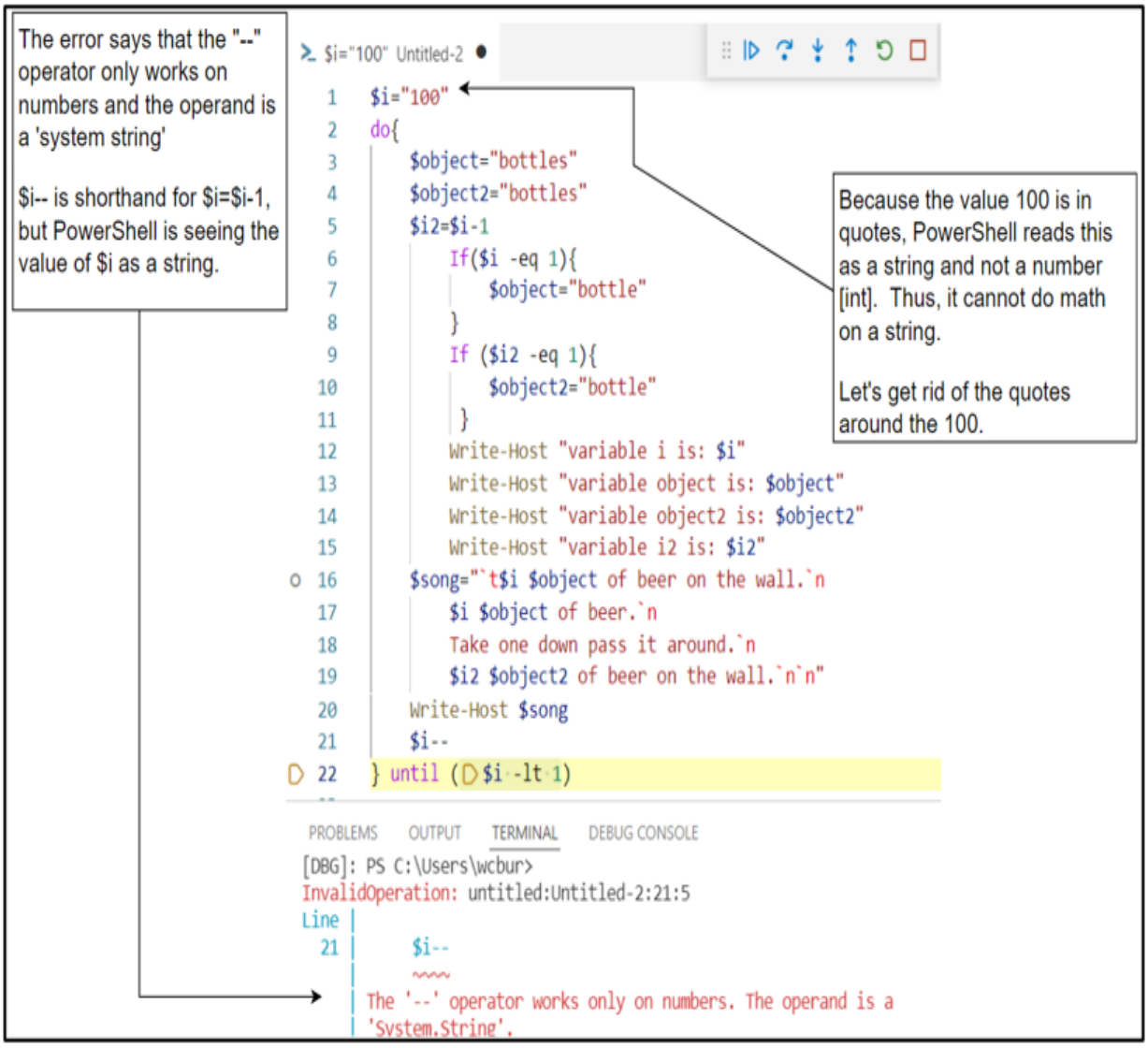


Figure 4.16 Final Issue!



The '--' operator only works on numbers (Figure 4.16). $i-- is shorthand for $i=$i-1. But since this is subtracting a value from a variable and replacing it, the variable needs to be a number.

Programming Shorthand

There are many common techniques that programmers run into over and over. To save themselves time, effort and typing a kind of shorthand was created for a lot of these very common things. Here are a few that will be useful for you to know at this stage.

++	Usage: $i++ The ++ shorthand takes the value of the variable and adds 1 to it. If you have a variable $i that currently has the value 15 in it, $i++ will make the new value of $i equal to 16. The more typing intensive way to write this is $i=$i +1. As you can see $i++ saves quite a bit of typing
–	Usage: $i – The – shorthand takes the value of the variable and subtracts 1 from it. If you had a variable $i that currently has the value 16 in it. $– will make the value of $i equal 15. The more typing intensive way to write this is $i=$i-1
+=	Usage: $array += $item The += shorthand takes the value of one variable and adds another variable to it. So if you have an array that has ('one','two','three') in it and you write $array += 'four'. The new value of the array would be ('one','two','three','four).

PowerShell is seeing the value as a System.String or (String) [Chapter 2]. Just like you can't subtract a number from a word (cat -1, for example, doesn't make any sense), you can't subtract an integer from a string. PowerShell is not seeing the value of $i as the number 100, but rather as the word "100".

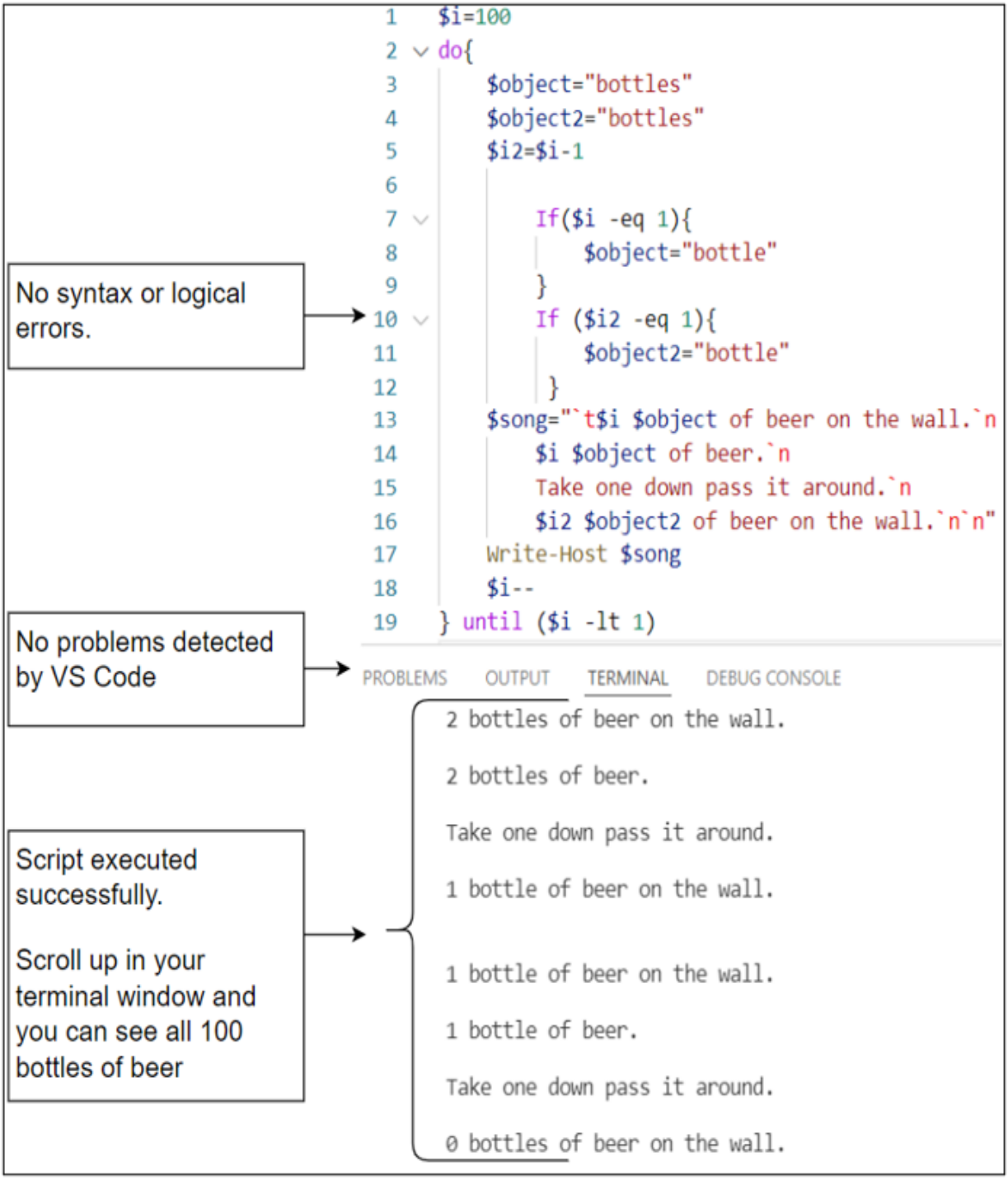
Strings are defined by putting them in quotes. We can remove the quotes from line 1 and PowerShell should now see the value of $i as the number 100.

With the loop issue resolved, we can remove the Write-Hosts displaying the starting values of the variables: i, i2, object, and object2.

```
$i=100
do{
    $object="bottles"
    $object2="bottles"
    $i2=$i-1

        If($i -eq 1){
            $object="bottle"
```

```
        }
        If ($i2 -eq 1){
            $object2="bottle"
         }
    $song="`t$i $object of beer on the wall.`n
        $i $object of beer.`n
        Take one down pass it around.`n
        $i2 $object2 of beer on the wall.`n`n"
    Write-Host $song
    $i--
} until ($i -lt 1)
```

Figure 4.17 Success!



Success! No syntax or logic issues (Figure 4.17). No problems identified by VS Code and all one hundred lines of the song has been printed to our output. Congratulations. If you followed the steps in this chapter you have some valuable experience and tools that you can use to help get scripts working that you create or find from other sources.

4.5 Summary

- When PowerShell displays an error, it will frequently provide significant help in resolving the issue. The error can be googled and often the output provides you a Microsoft Knowledge link that will give details on the specific function that is causing the script to terminate.
- Scripts can contain both Syntax errors and logical issues that may cause the script to terminate unexpectedly (crash) or simply not produce the results you are expecting.
- VS Code will highlight many Syntax issues with *red squiggly lines* to help you identify where you have syntax issues within your script.
- If you're using Microsoft VS Code as your IDE, there is an integrated syntax checker that can be accessed from the "Problems" tab of the output screen. This checker can quickly identify your syntax issues within the script.
- Logical scripting issues are when an issue with the script causes it to operate incorrectly without causing it to terminate unexpectedly.
- When troubleshooting logical errors, you can utilize *Breakspoints* to step through the code one section at a time until you find the point where the script behaves differently than what you expected.
- You can check the value of variables and inputs by utilizing the write-host cmdlet to make sure the value matches the value you expect.
- Most variables are stored in the PowerShell session they were initialized in and often stay resident in the session after the script terminates. When debugging you clear the value of your variables before you use them to make sure the value is of the variable is set the way you expect.

5 The power unlock

This chapter covers:

- Storing large amounts of objects and referencing them individually or in groups.
- Performing actions to any number of objects transforming dozens or thousands of tasks to a single click.
- Unlocking Specific locked user accounts without having to search Active Directory.
- Unlocking your whole domain with three lines of code.

If there's one constant in the world of IT it's the fact that users will lock themselves out and come to you, the admin, to unlock them. Typically, this involves opening up Active Directory Users and Computers and then searching for the user, opening their account clicking on a tab and checking a box.

It doesn't take up a ton of time. But it is something that can be made much easier with PowerShell. Now, imagine, for a moment that you have a network issue in one of your remote locations. Some service has stopped and all of your users in a particular geographic area are locked out.

At this point you can be reactive and wait for tickets or phone calls to roll in and unlock the users one by one. Or with the use of this script, you can see with a single click that there is a significant number of users from any given network location that are locked out.

This simple inclusion in a script that you'd be using to make your life easier can take you from reactive to IT superstar.

5.1 What Does the Script do?

To unlock an account in active directory you would typically need to launch Active Directory Users and Computers. Then navigate to the user in the OU

structure either via exploring the OUs or utilizing the search function. Once you have the user selected, you open the user properties explore to the "Account" tab and check the *"Unlock account. This account is currently locked out on this Active Directory Domain Controller." Check box and click OK.*

This process is okay if you have one or two accounts you might want to unlock, but it's tedious and reactive. You don't have a good way of proactively seeing who on your domain is currently locked out, so it's difficult to pick out trends.

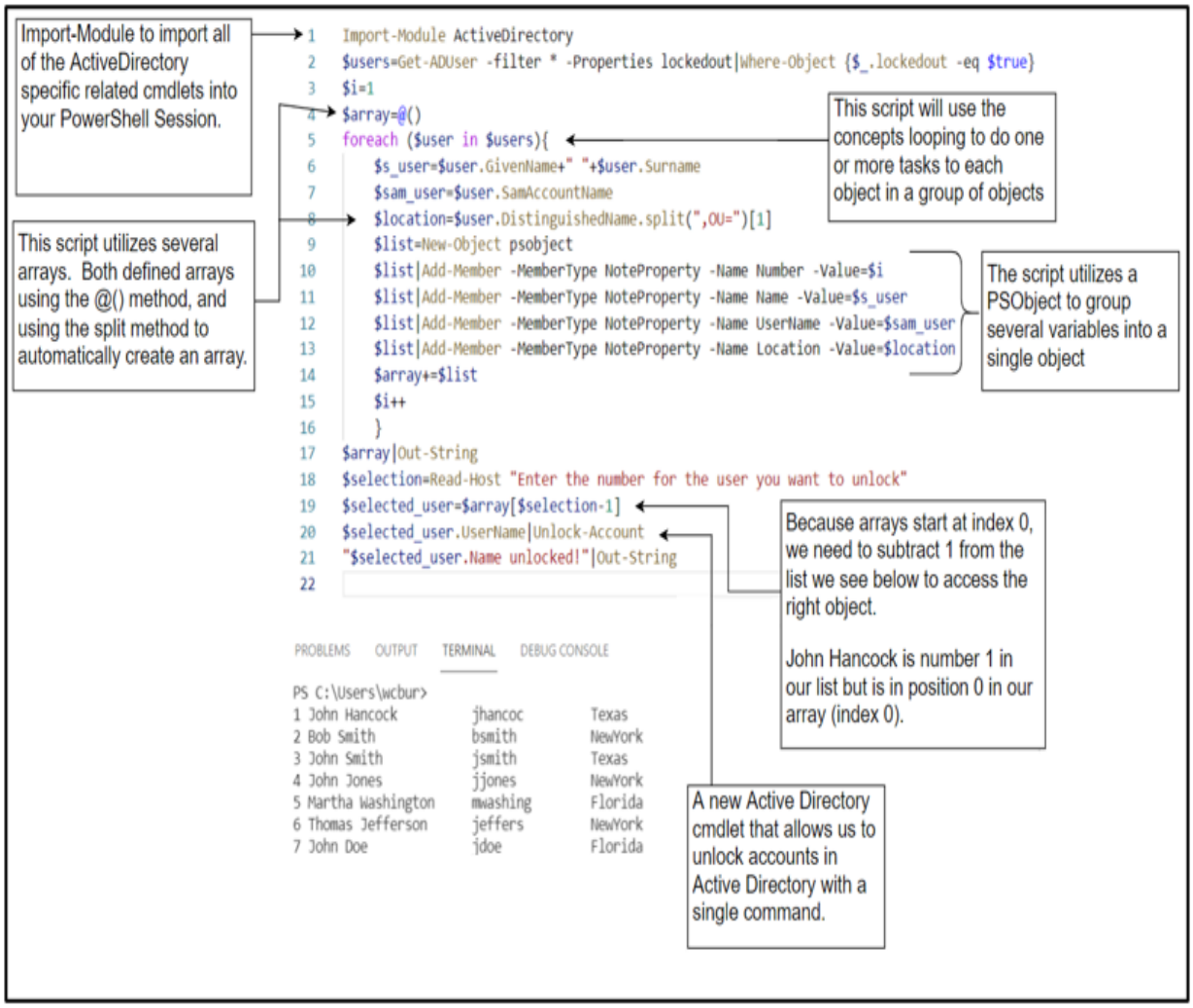
The code below will:

- Find all of the users in your whole domain who are currently locked
- Pull each user's name, username, and location into a numbered list
- Prompt the user which user they'd like to unlock
- Unlock the corresponding user.
- Confirm the user is unlocked.

5.2 Project Code

Instead of all of that manual and tedious clicking and typing we can simply let PowerShell do the work for us. The code to do this is listed below (Figure 5.1).

Figure 5.1 Power User Unlock Code



5.3 Arrays: a great place to keep your stuff

Up until now, our scripts have been fairly basic. We have prompted users for names and used those names to perform functions without much logic or repetition. This script is the first we will introduce that contains loops, arrays and conditional logic to actually use the power of the language to become more functional than a simple command line. We'll be building on this skillset and tools to perform more and more complex tasks that will make automating your network simple and fast!

Arrays are part of what makes PowerShell more flexible and powerful than many other scripting languages. We've already discussed variables and how

useful they can be for storing things like, strings, integers, and objects for later use.

Arrays do the same thing. But unlike variables which store only one value. Arrays are capable of storing many different values at the same time!

Variables vs Arrays

In chapter 2 we first were introduced to the concept of a variable. A specific place in memory where our script could store things for later access. We compared this to a shoe box. A place that you could put something and later get it out when it was useful.

If a variable is a shoebox, think of an array as a whole wall of shoe storage (Figure 5.2).

Figure 5.2 Think of an Array like a collection of Shoe Boxes

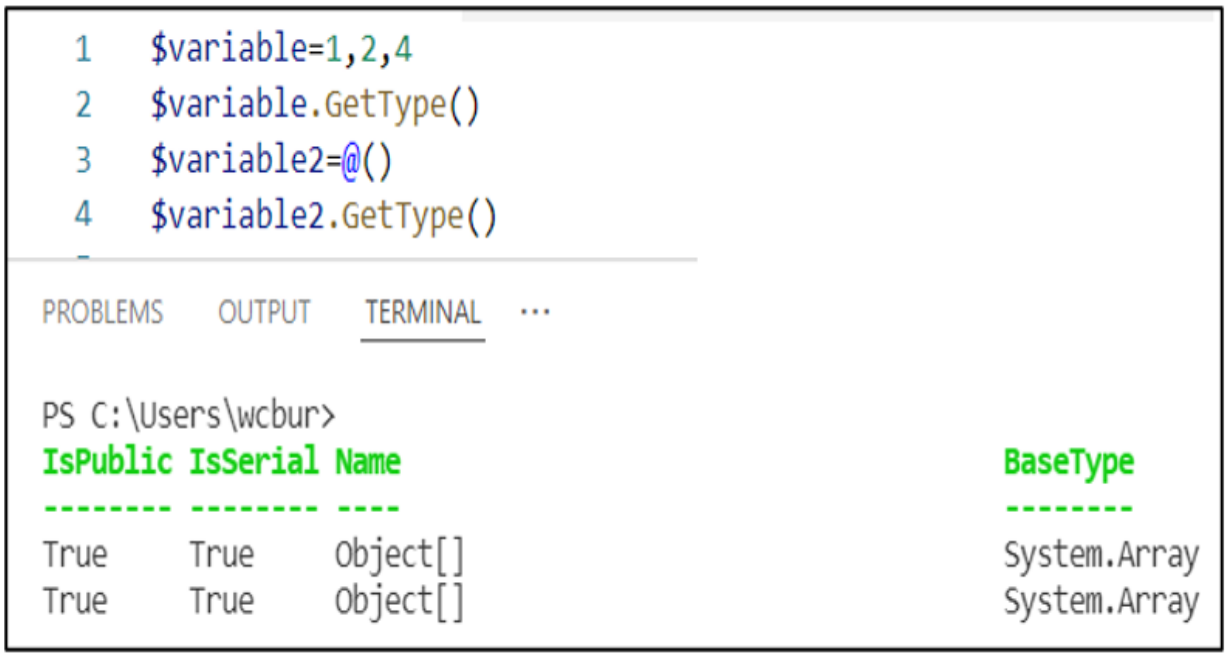
Calling an Array

Just like in PowerShell you denote a variable by starting with the dollar sign (`$`), arrays are instantiated with the @ and enclosed in parentheses. `@()`. Each object inside is differentiated by a comma.

Two Methods for Creating Arrays

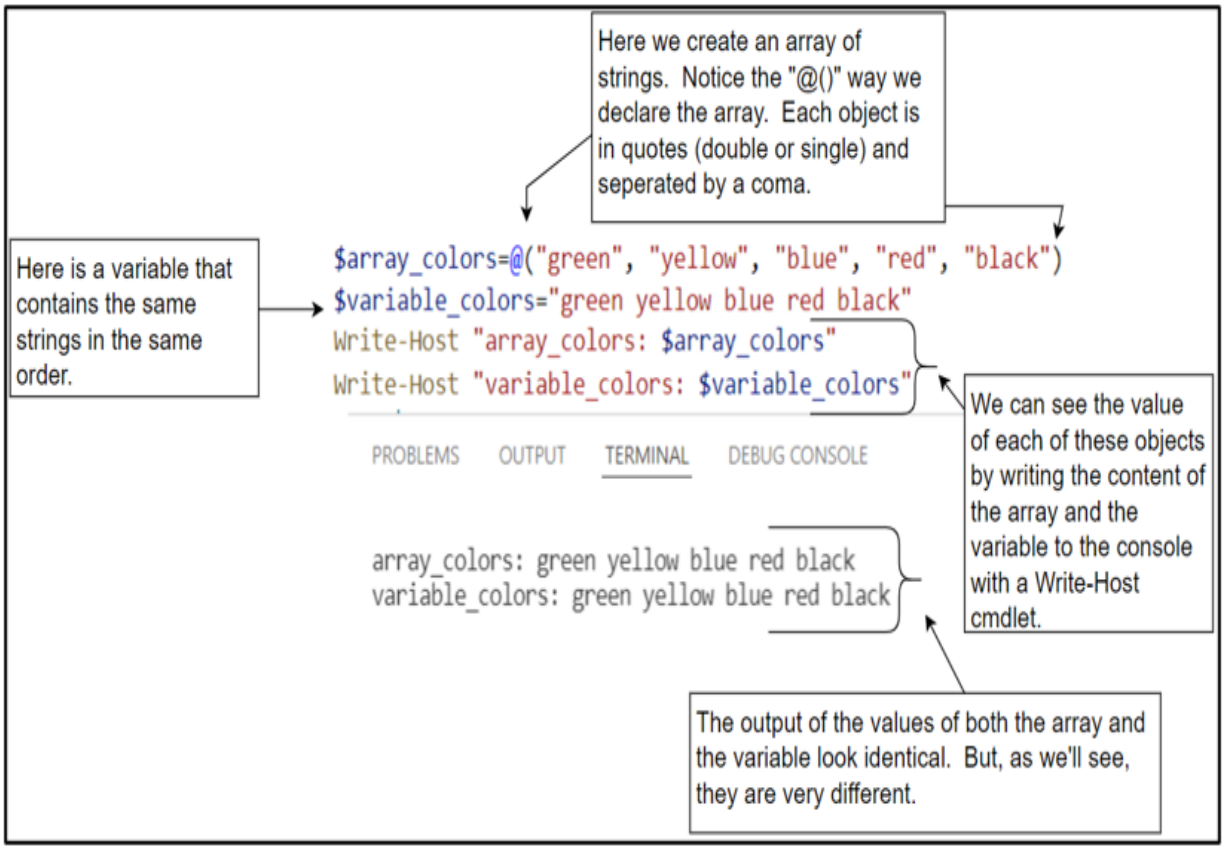
You can use the @() technique to force an array object. However, arrays can be created in an even simpler way. When you separate the values of a normal variable with a comma, PowerShell will treat the values stored in that variable as an array. You can use the GetType method to return the data type of the variable. In this case we can validate that both instantiation methods result in PowerShell treating the variables as array objects (Figure 5.3).

Figure 5.3 Creating Arrays.



On the surface a variable and an array may be hard to distinguish, but it will quickly become evident how different they are (Figure 5.4).

Figure 5.4 Arrays and Variables look similar at first.

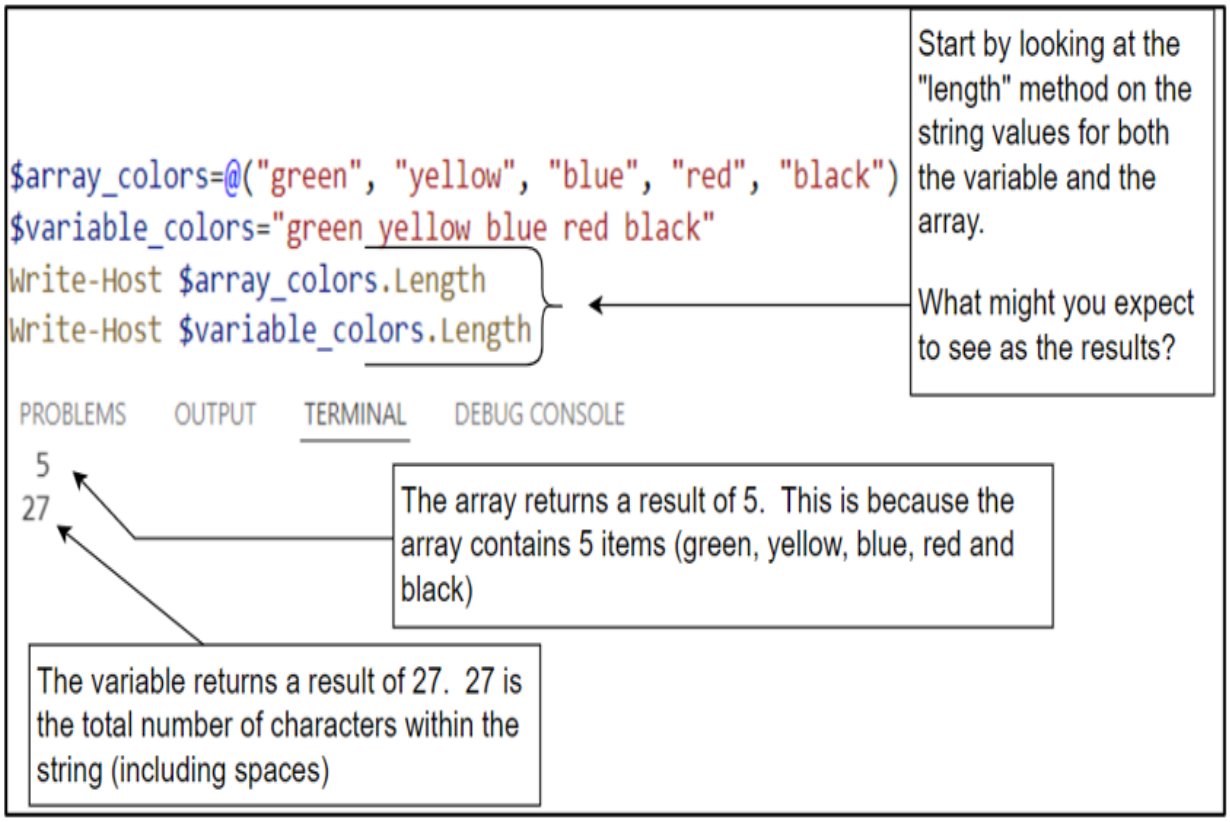



```
$array_colors=@("green", "yellow", "blue", "red", "black")
$variable_colors="green yellow blue red black"
Write-Host "array_colors: $array_colors"
Write-Host "variable_colors: $variable_colors"
```

We can assign the same values to the array and variable above. When we write them out the output appears to be identical.

However, when we try to use the data within the array and variable they behave very differently (Figure 5.5).

Figure 5.5 Although they look similar PowerShell treats Arrays and Variables very different



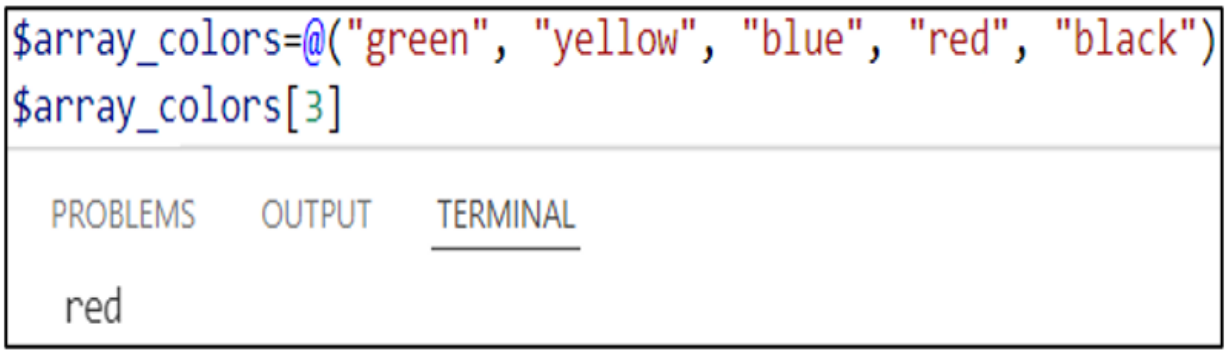
This distinction makes the data stored in the array much more available to us for manipulation.

Referencing Array Objects

But how do we access the values in the array? Like the shoebox example, let's say you wanted to get a specific pair of shoes from the rack full of shoeboxes. How can we make sure that we get exactly the box that we want? The easy way is to number the boxes, then when we want the shoes in box four, for example, we can ask for the shoes specifically in box four. Fortunately, arrays work in a very similar way. When an array is created each object in the array is numbered as well. These numbers are known as indexes. We can reference any index of an array by putting the index number inside square brackets: [some number]. This tells PowerShell exactly which index to retrieve and will return the value of the data in that index.

Just like in the above example, if we wanted to retrieve the value in the fourth index of an array, we would type the name of the array and then the index number in square brackets (Figure 5.6).

Figure 5.6 Accessing an array.



This would retrieve the fourth index of the array, in our case the string ‘red’.

Now, you might be wondering, if we wanted the value of the fourth index of the array, why did we use:

```
$array_colors[3]
```

It is common for an array in a programming language to start with 0 and count up from there. In the early days of computers and computer languages calculations were a valuable commodity. Starting at zero and counting upwards is more efficient. This is known as zero based indexing.

Why 1 based indexing is less efficient

To calculate the nth term as a+(n-1)*d

a= first term

n= index

d= common difference

For simplicity, let’s look at an array that increases by 5.

$array=[5,10,15,20,25,30,35,40]

If we wanted the 3rd index of the array, we would need to use the formula:

a+(n-1)*d

a=5 (the first index)

n=3 (the index we want)

d=5 (the difference)

5+(3-1)*5=15 which is correct!

However, if we used a 0-based index our formula would be a+n*d

Again, if we wanted the 3rd element of our array it would be

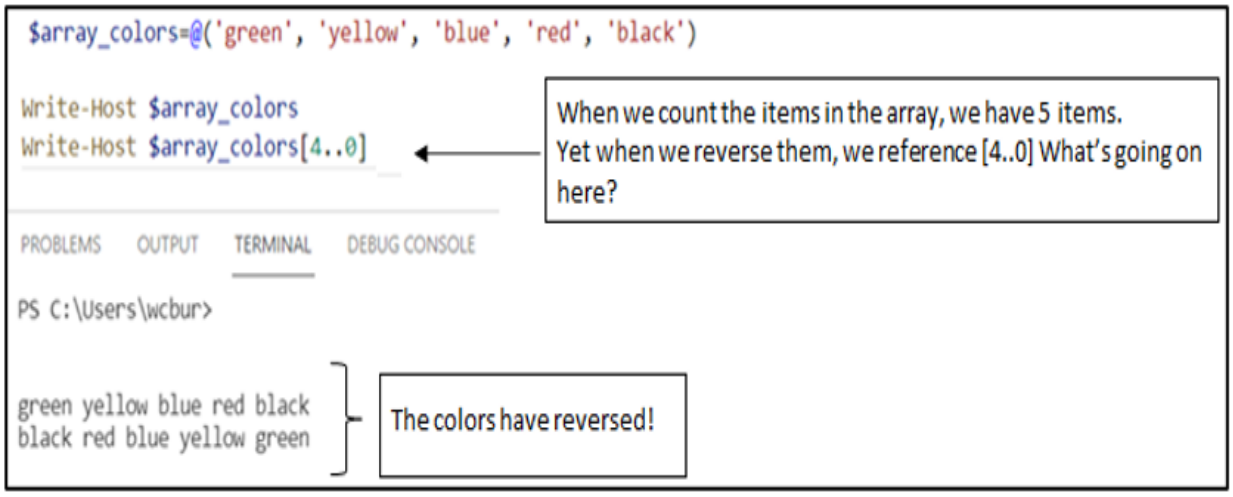
5+2*5=15

We got the same answer, but with zero based indexing we didn't have to subtract 1 from n. For the computer this is a calculation saved. Today's modern computers can execute between 150,000,000 to 200,000,000 calculations per second. The IBM 603 developed in 1946, however, could make as many as 6,000 calculations per hour. So, you can see why in the early days of computers you would want to save as many calculations as you could.

Referencing the values of the indexes of the array makes it far easier to access and re-arrange those values. Let's say for, example, we wanted to change the order of the colors in our output. If we wanted to reverse the colors in the array it's simple (Figure 5.6).

```
$array_colors=@('green', 'yellow', 'blue', 'red', 'black')
Write-Host $array_colors
Write-Host $array_colors[4..0]
```

Figure 5.7 Manipulating values in arrays.

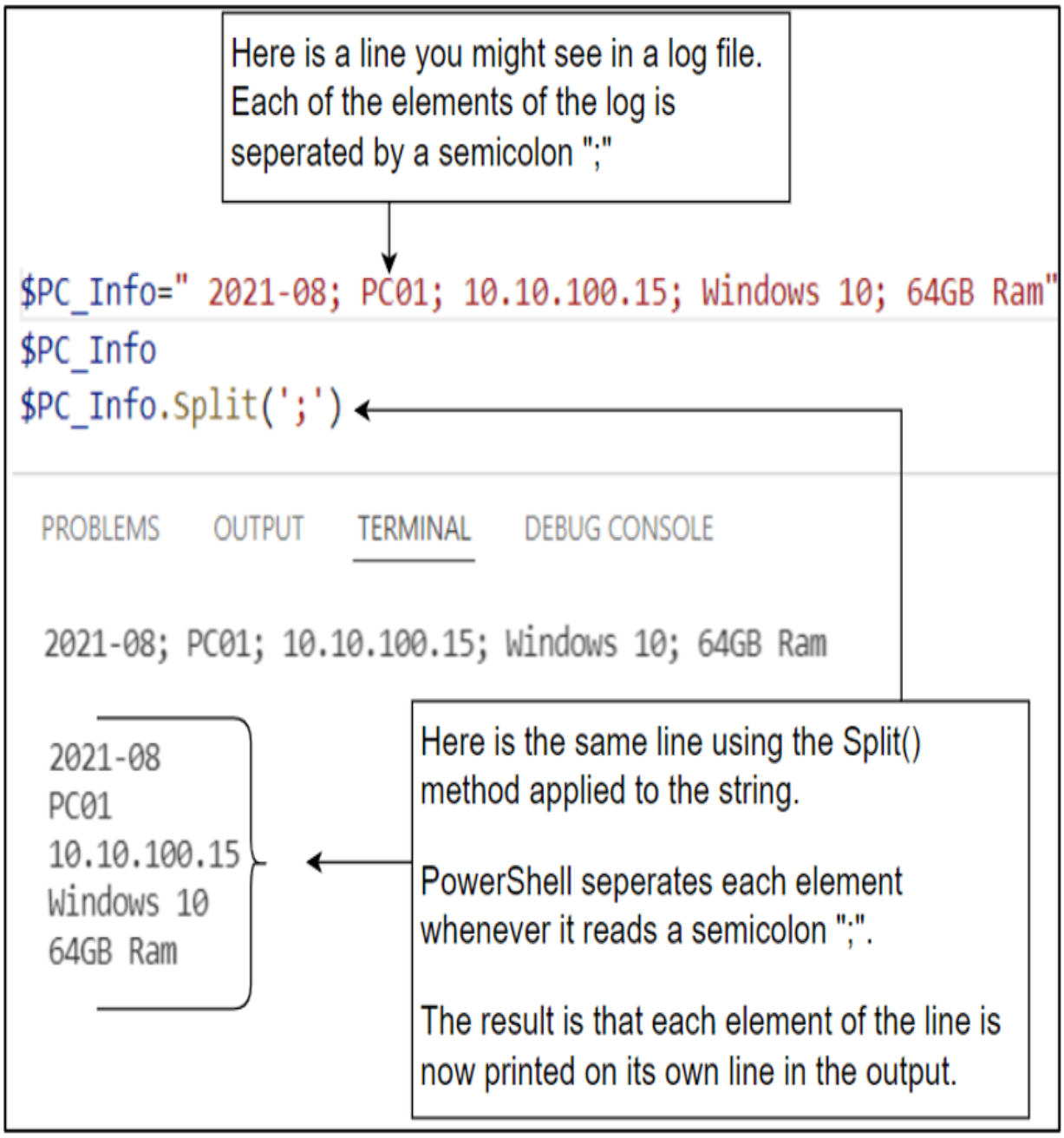


As you can see in the image when we look at the number of items in the `$array_colors` the results are 5. However, when we reference the array in line 7, we use `[4..0]`. This is shorthand for "everything between the numbers 4 and 0 (4, 3, 2, 1, 0).

Splits

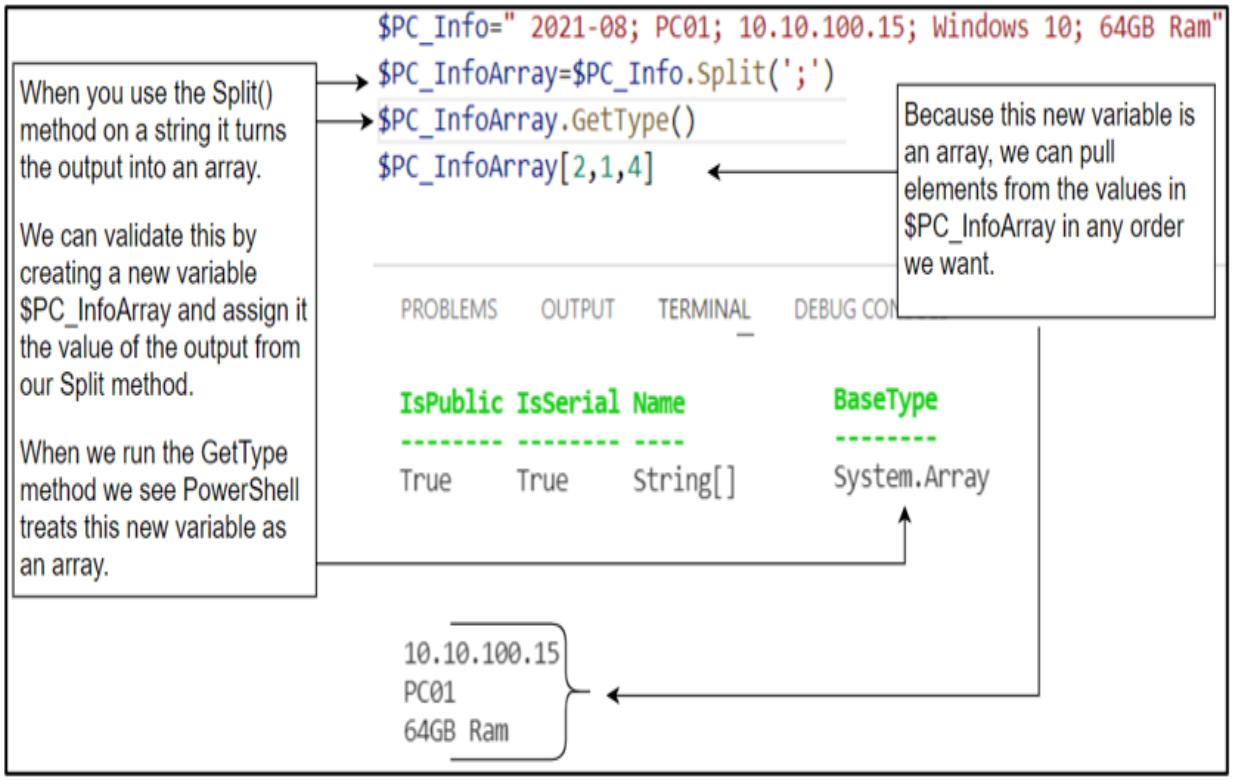
There are times you want to break up a string and select only the parts or part you want. If there is a specific pattern to the string you can use the `split()` method to basically cut the string into parts (Figure 5.7).

Figure 5.8 Using the Split Method to create an array.



Each part is then stored in an array, and you can use the same techniques we just learned to access the array as a whole or any specific part we want (Figure 5.8).

Figure 5.9 Accessing the elements of an array.



5.4 Loops

There are many times in scripting or any kind of coding where you want to do something specific to many different items in a list. For example, say you get a report from a team member that lists all of your PC items and the last user logged in. The list when exported looks something like this:

PC01:gwashington; PC02:jadams; PC03:tjefferson; PC04:jmaddison; PC05:jmonroe;

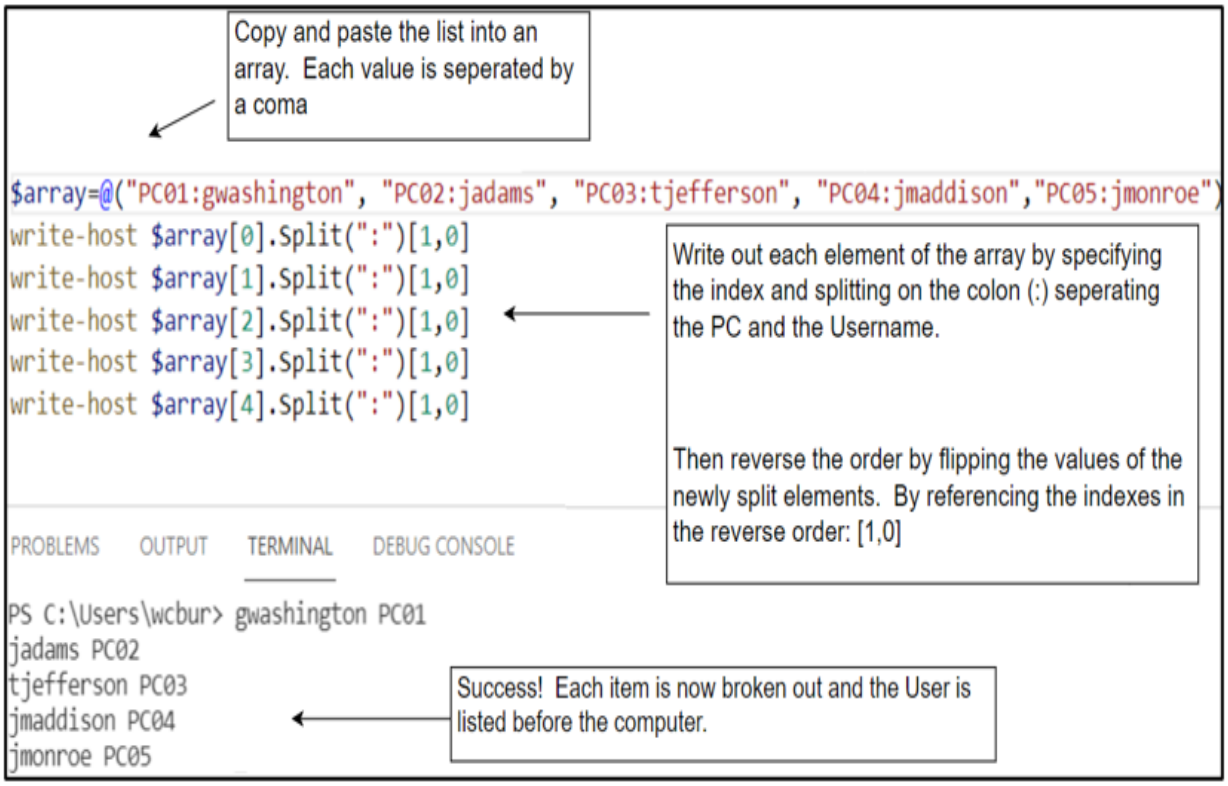
However, your boss doesn't like the output. Your boss would prefer that the report listed the user logged in followed by the computer that they were logged into. At this point, you can try to regenerate the list, which may be difficult or time consuming or we can use PowerShell to do the work for us.

Here we can import the list into an array. Since each one of these items is in quotes it indicates to PowerShell that these objects should be strings. So instead of having one value in a variable, we have five string values in an array. Each one of these values is a separate element of the array. Since

they are string objects we can use the split() method and then split on the colon that divides each of the PCs and users. This will automatically store the new elements in a new array. We can then flip the order of the indexes of this new array and write this to our console with a write-host (Figure 5.9).

```
$array=@("PC01:gwashington","PC02:jadams","PC03:tjefferson","PC0
4:jmaddison","PC05:jmonroe")
write-host $array[0].Split(":")[1,0]
write-host $array[1].Split(":")[1,0]
write-host $array[2].Split(":")[1,0]
write-host $array[3].Split(":")[1,0]
write-host $array[4].Split(":")[1,0]
```

Figure 5.10 Modifying individual elements of and array (the hard way)



This code works. But it was only 5 elements. Imagine if there were on hundred, or a million, that's a lot of typing. There must be an easier way!

This is where loops come in. A loop sets up a section of our code that executes over and over until a condition is met. It can run once or an infinite number of times, until the condition is met. What if, instead of listing out

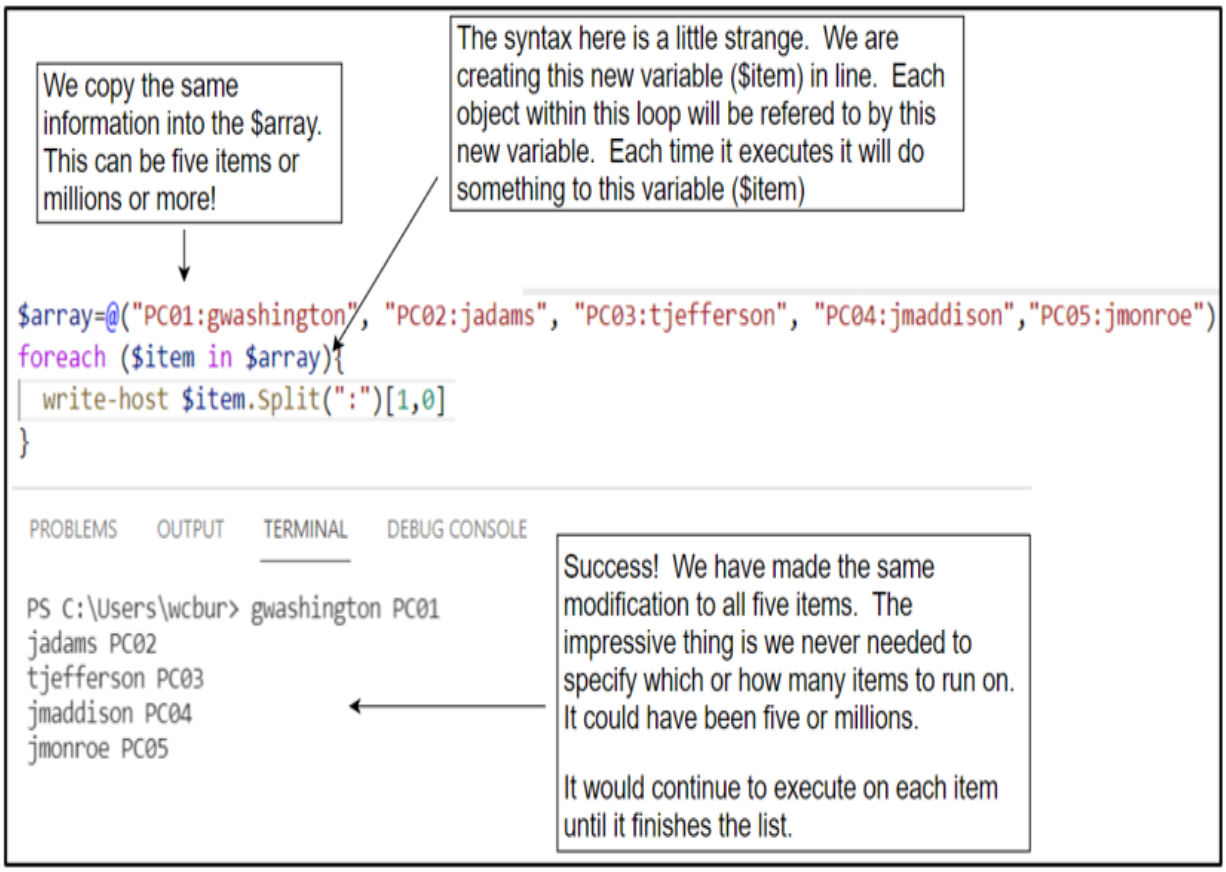
each of the elements in our $items we told PowerShell to do the same thing for each one until there were no more left? That is exactly what a for loop does!

For Loops

The first type of loop we're going to explore is called the for loop. Where for each object in the list we are going to do the same things to it (Figure 5.10).

```
$array=@("PC01:gwashington","PC02:jadams","PC03:tjefferson","PC0
4:jmaddison","PC05:jmonroe")
foreach ($item in $array){
    write-host $item.split(":")[1,0]
}
```

Figure 5.11 Modifying the array the easy way



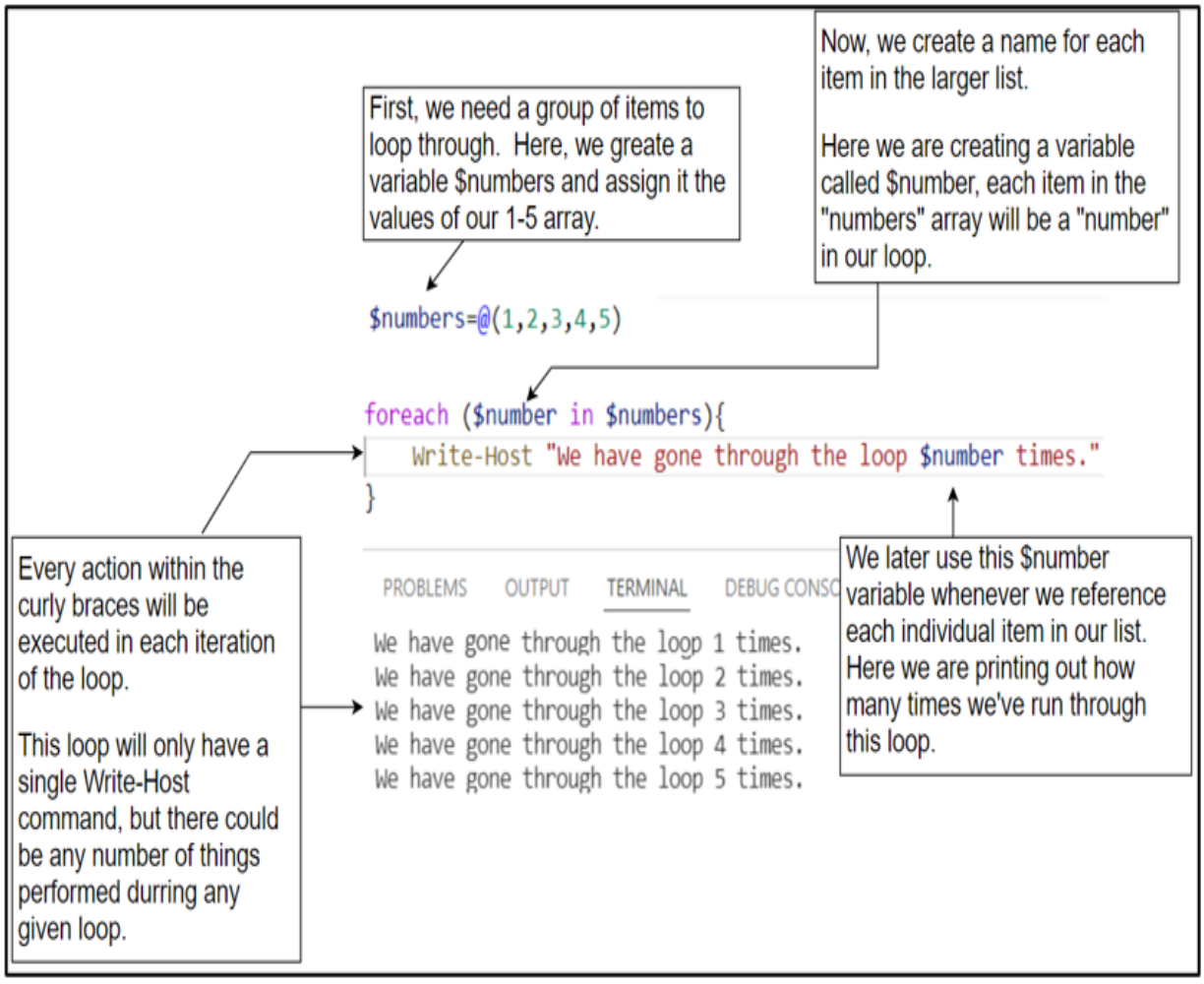
A for loop is defined with the code:

```
Foreach ($new_variable in $existing_list){
    Do stuff to each $new_variable
}
```

The structure of a for loop (Figure 5.11) should include at least these three things to make it work correctly.

1. The group of variables to loop through
2. The variable assigned for each item in the group of variables
3. The curly braces that make up the boundaries of the loop.

Figure 5.12 Structure of a foreach loop



There are many types of loops, all of which essentially do the same thing, by performing one or more actions to one or more targets. We'll cover more of the different loop types in future chapters. The difference between loops is largely how the loop determines if and how many loops need to be done. The foreach loop does the same action to each target until there are no more targets. Other loops, like do while and do until loops, use conditional logic to determine if the loop still needs to run or if it has already met the condition to stop the loop.

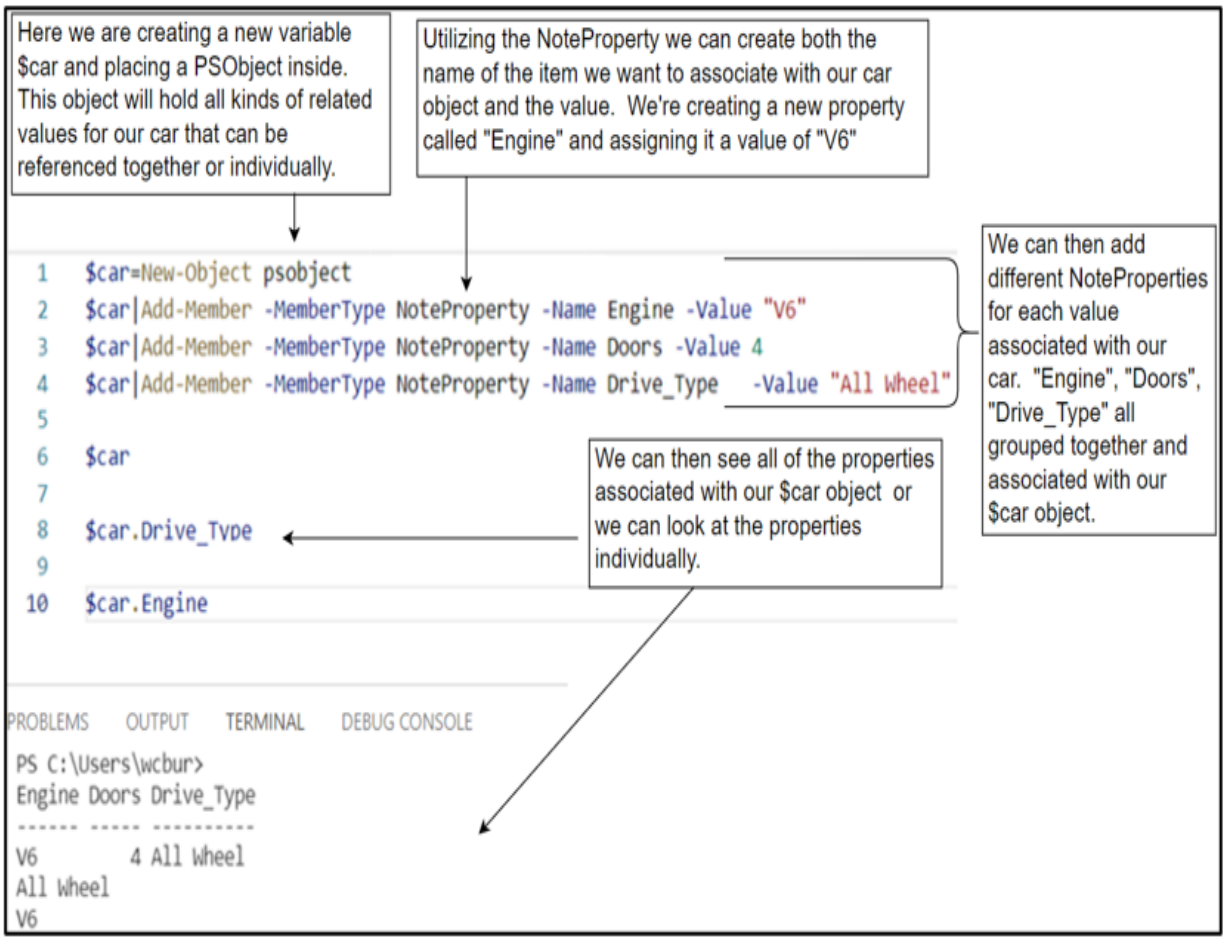
5.5 PSObjects

So far, we have covered a number of ways to store information. We can store information in a variable and in an array. But what if we need to store multiple pieces of information about a single type of object? In the Tiny PowerShell Project code we are looking at a user. Then we are pulling the user's SamAccount name, the user's full name, and the user's location from Active Directory. We can pull all of these into variables; we can even store all of this information in an array. But how do we do this if we want to make sure that all of the values are associated to each other?

By utilizing a PSObject we can take a group of objects and collate them into a single object. Once we have created the PSObject, we can use the cmdlet called "Add-Member" to add a "NoteProperty" to the item. A NoteProperty is a generic property that PowerShell can create and assign to an object. When you create a NoteProperty you assign it a name and a value and PowerShell assigns that property to the object.

If we had a group of related variables we wanted to group together as a single variable, then we can then add all of the Member Items to a PSObject. This creates a single item that keeps track of all of the related properties of the object (Figure 5.12).

Figure 5.13 Add-Member cmdlet use



5.6 Unlock-ADAccount

As already mentioned, PowerShell is made up of Command-lets. When we import the ActiveDirectory Module the native cmdlets associated with Active Directory are included. We've already seen cmdlets like Get-ADUser, but there are so many functions that can be performed within Active Directory, and most of these functions have an associated cmdlet.

To unlock a user in Active Directory Users and Computers (ADUC), you would need to:

- Explore to or Search for the User
- Right click on the user account
- Select Properties
- Navigate to the "Accounts" tab.

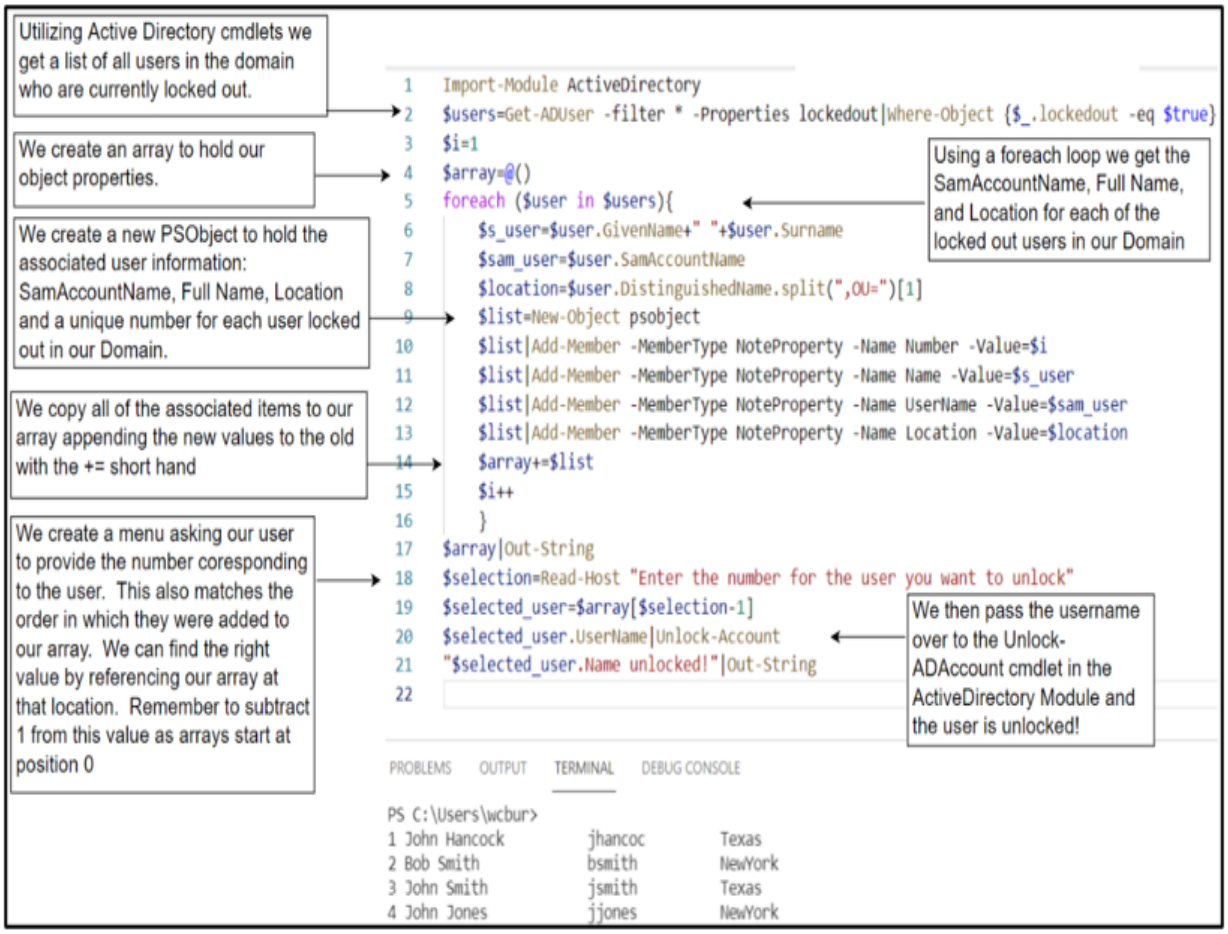
- Select the “Unlock Account” check box
- Press OK.

PowerShell gives us a single cmdlet that does all that work for us.

```
Unlock-ADAccount jdoe
```

This single command performs all these steps and unlocks jdoe’s account (Figure 5.13).

Figure 5.14 Tiny PowerShell Project code review.



5.7 Try this too

Locking out a user’s account is a security feature. It is the primary defense against a bad actor using a brute force attack to gain access to your network.

For this reason, I don't suggest simply unlocking all accounts when they become locked. This is the perfect situation for a brute force attacker.

However, there are times, when there is a legitimate reason to unlock all accounts; for example, an Exchange Server issue might cause a large number of accounts to be locked. Instead of unlocking all these accounts manually, or even with the script we developed in this chapter, you can unlock them all at once. Sometimes the efficiency of unlocking a large number of locked accounts outweighs the potential security risk of unlocking all of your accounts with a single click.

It is, after all, for this very power and flexibility we've turned to PowerShell in the first place.

5.7.1 Unlock All Locked Accounts.

With three lines of PowerShell Code, you can unlock all of your locked users with a click.

```
$users=Get-ADUser -filter * -Properties lockedout
|Where-Object {$_.lockedout -eq $true}
Foreach ($user in $users){
    $user.SamAccountName|Unlock-ADAccount
}
```

This task manually, could have taken an entire team of people hours to accomplish. Congratulations, you're a superstar.

5.8 Summary

- There are many ways to store information with PowerShell. Arrays are powerful data structures designed to hold many elements of information and treat each element as a separate data piece.
- Referencing an array is as easy as calling the whole array, like you would a variable, or by referencing a specific element or multiple elements from the array by utilizing the square braces.
- Array indexes start at 0 and the indexes count up from there. The first value of the first element of the array, in PowerShell, can be called by

accessing index 0.

- Loops are ways your program or script can perform the same action or actions on multiple targets until an end condition is met.
- Loops can run zero to an infinite number of times depending on the conditions you set for the start and end of the loop.
- The foreach loop will perform the same action or actions on each element in a list until it performs the tasks on all of the elements of the list, at which point the loop terminates.
- PSObjects are objects that can be used to help you keep groups of related information together.
- PSObjects can use the Add-Member cmdlet to add custom properties to the object known as NoteProperties.
- Using Active Directory Cmdlets, arrays, PSObjects and Read-Host menus you can create a script that can unlock any locked user on your domain with simple menu selection.

6 Manage groups like a boss

This chapter covers:

- The basics of utilizing conditional logic in our scripts
- Identifying the Active Directory Groups any User in our Domain is a member of
- Using an existing user as template to copy group membership to anyone with a single script

It's common in today's environments to manage file access with the concepts of least privilege and Role Based Access Controls (RBAC). This prevents anyone without a need to access specific company data from accessing the data. This is exactly what Active Directory groups were created for.

But managing groups and making sure that new users have all the file access they need can be cumbersome at best and a true nightmare at worst. This Tiny PowerShell Project will throw admins a lifeline in this regard.

Traditionally, when you want to create a new user in a department you look for the groups that they will need to be a member of. Few things are more frustrating to users and their managers than waiting days to get their new employee a login then waiting weeks before that employee can actually start doing what they were hired to do.

This usually entails:

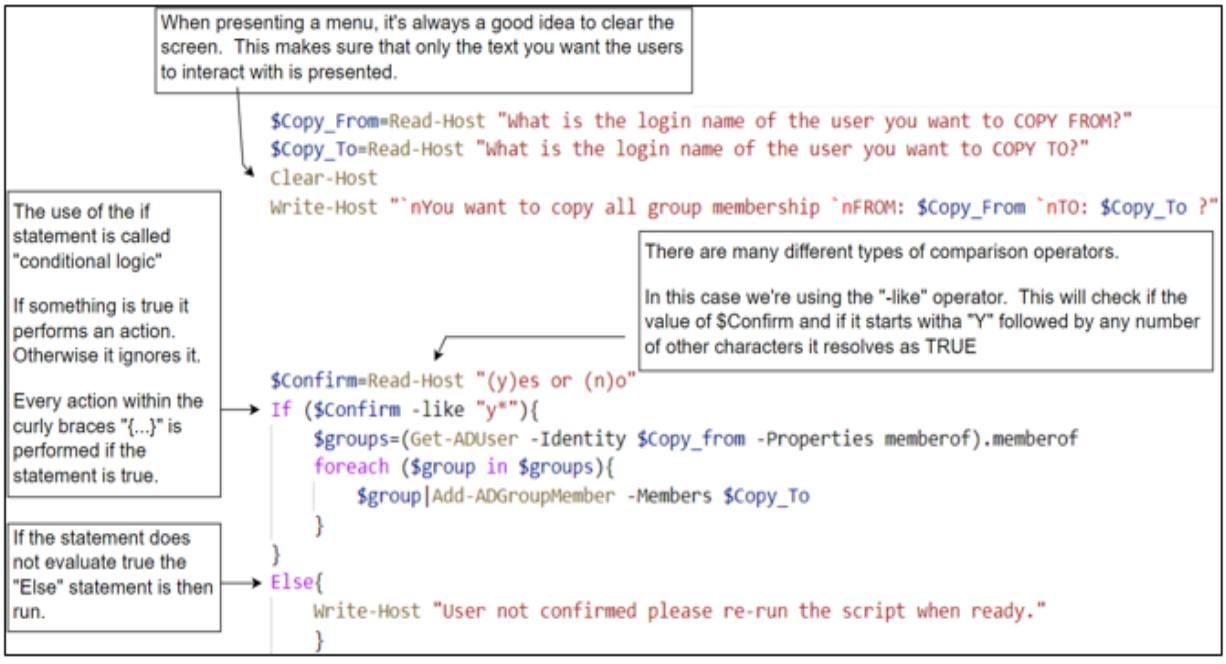
- A system administrator finds a user who is in that department
- Open Active Directory Users and Computers (ADUC)
- Search Active Directory for the user
- Right click on the user
- Select Properties
- Go to the "Member Of" tab
- Double Click on a Group in the you wish to add a new user to.
- Select the Members tab

- Click on the Add button
- Type the new user's name and click OK
- Then repeat this for each group you want to add.

This script (Figure 6.1):

- Prompts the user for the login name of the user they wish to copy from
- Prompts the user for the login name of the user they wish to copy to
- Confirms the selections are correct (this is not required for the cmdlets to work, but since we're dealing with security relevant objects like Security Groups it's a good idea to make sure there's a chance to catch any mental flubs)
- Looks up the template user in Active Directory and copies all of the groups they are a member of into an array so we can import these same groups to the target user.
- Copies groups in the array to the target user.

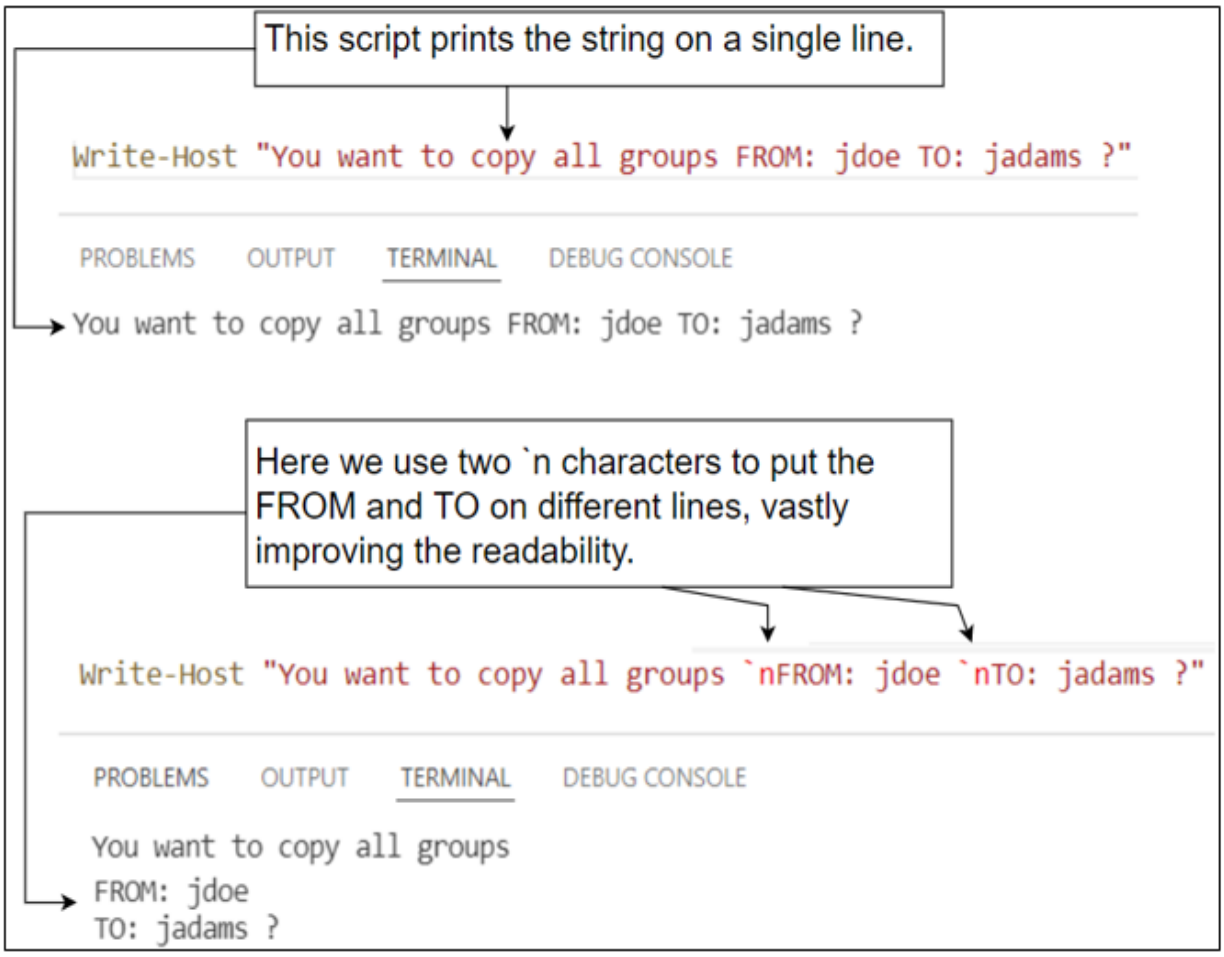
Figure 6.1 Tiny PowerShell project code.



Reminder: New Lines in Strings

In several places in this chapter's script you see the `n within the string. This is PowerShell shorthand for a new line. This makes the output much easier to read (Figure 6.2).

Figure 6.2 How and why to use new lines within strings.



6.1 Using Conditional Logic

Arguably the most important skill you can learn while coding or scripting is a firm grasp of Conditional Logic. Conditional logic looks at some input, variable or result and makes a decision depending upon the value.

Humans use conditional logic all the time, it's so ingrained in our everyday behavior we rarely even realize we're using it.

Imagine we were teaching a computer to drive a car. We can tell the car to:

- Drive
- Stop
- Turn

If we wanted the car to turn right at the second intersection. We might give it instructions like

- Drive to the first intersection
- Drive to the second intersection
- Turn Right

However, if we gave the car these instructions, we might expect a lot of accidents, because the car doesn't know how to interact with intersections. If there were stop lights at these intersections the car would drive through them regardless of the color of the light.

This is because the instructions we gave it were to drive to the first intersection and then drive to the second intersection. There was no conditional logic on how to treat a stop light. So if the light is green, the car drives to the next intersection. If the light is red, it still drives to the next intersection. You can quickly see why this might be a problem.

Much like teaching a car to choose the correct actions for different traffic signals, you might want a way to treat inputs, outputs or objects differently from each other. You wouldn't treat the local administrator account the same way you would a typical user account. You wouldn't treat a Domain Controller the same way you would a workstation.

But with the use of conditional logic, we can have the script begin to make decisions on how it behaves based upon the inputs, outputs or objects it interacts with.

Where-Object

We have in previous chapters already used some comparison operators to filter data. The Where-Object that we use to find specific User Accounts in

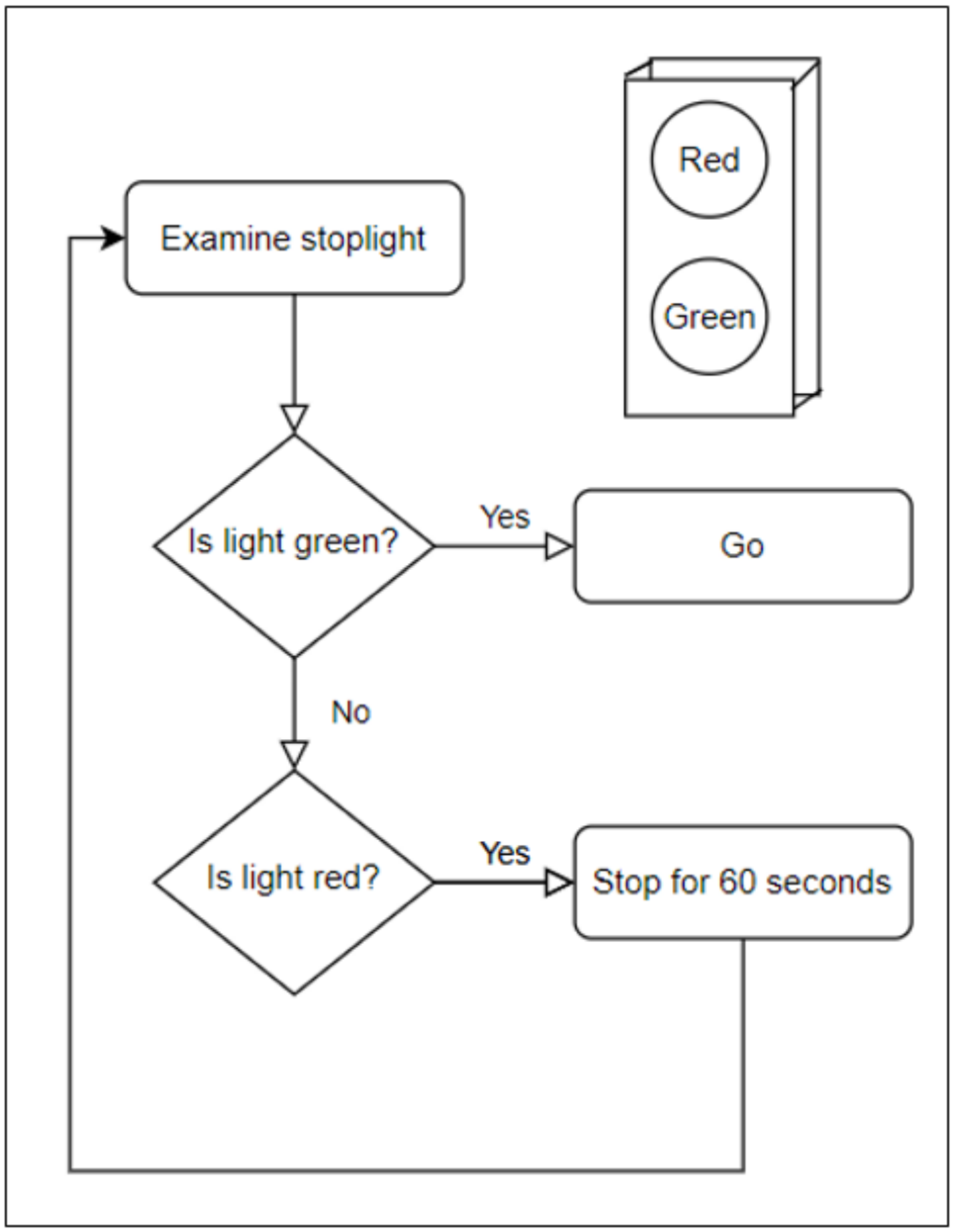
Chapter 3 or Locked Users in Chapter 5 utilizes comparison operators to only display a subset of the results based upon the conditions we set in it. This is very useful, but we can also use these same comparison operators to produce conditional logic. Conditional Logic allows us so much more power and flexibility than simply filtering.

Going back to the car example: What if we gave the car some ability to understand conditional logic? Not only would we give it the basic commands above, but we would give it two more abilities.

- Drive
- Stop
- Turn
- If the light is green, go
- If the light is red, stop

Now as the car drives to the first intersection it's able to act differently depending upon the color of the light. If the light is green it will go, if the light is red it will stop (Figure 6.3).

Figure 6.3 Conditional logic example



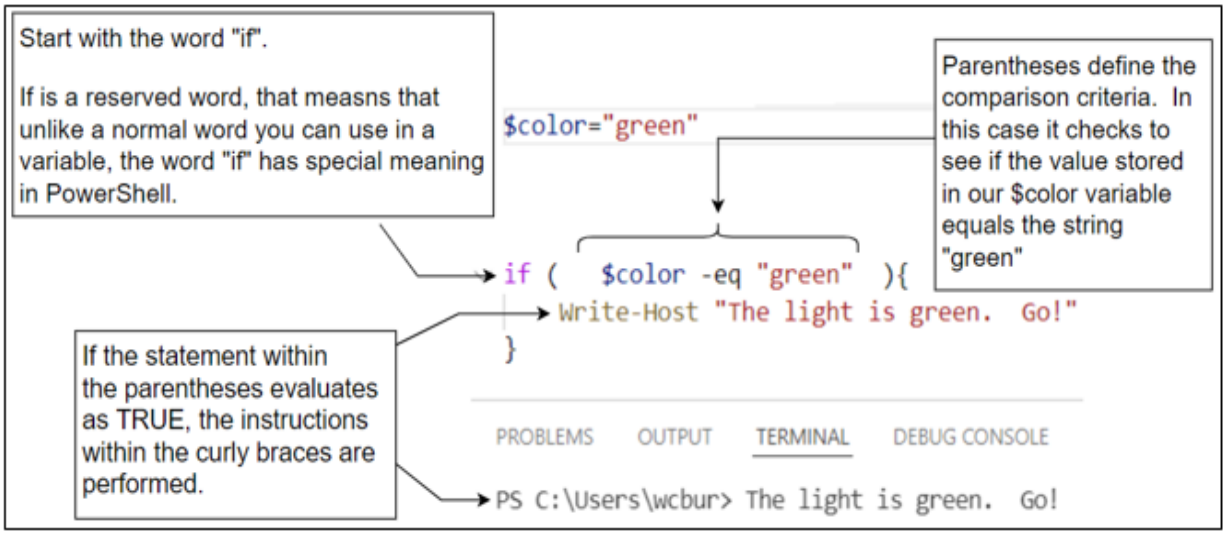
Just like the car without traffic light awareness, when you execute simple scripts, PowerShell starts at the first line and executes it and every sequential line of code in order until it reaches the end. But, with the use of conditional logic we can now start to branch that code and have it perform different lines of code for some conditions and others for the rest.

6.1.1 If

The most basic conditional logic we can use is known as the “if statement”. If a condition is met, then perform an action. In the car example we use this exact statement “If the light is green, go.” You can use the same statement in your code.

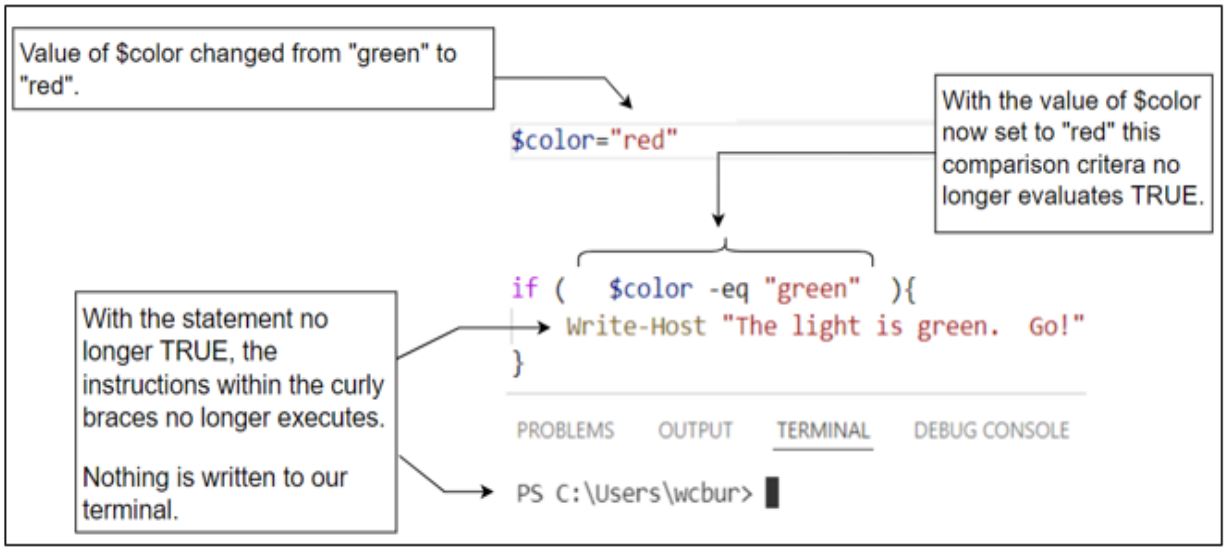
To use the `if` statement, you use the structure shown in Figure 6.4. Start the line with if followed by two parentheses. The statement you are comparing goes into the parenthesis. If the statement is true anything within the following curly braces will be executed.

Figure 6.4 If example: True



What happens if we modify the value of $color so it no longer evaluates as true? As you can see below (Figure 6.5), once this is no longer true, the code within the If statement no longer executes!

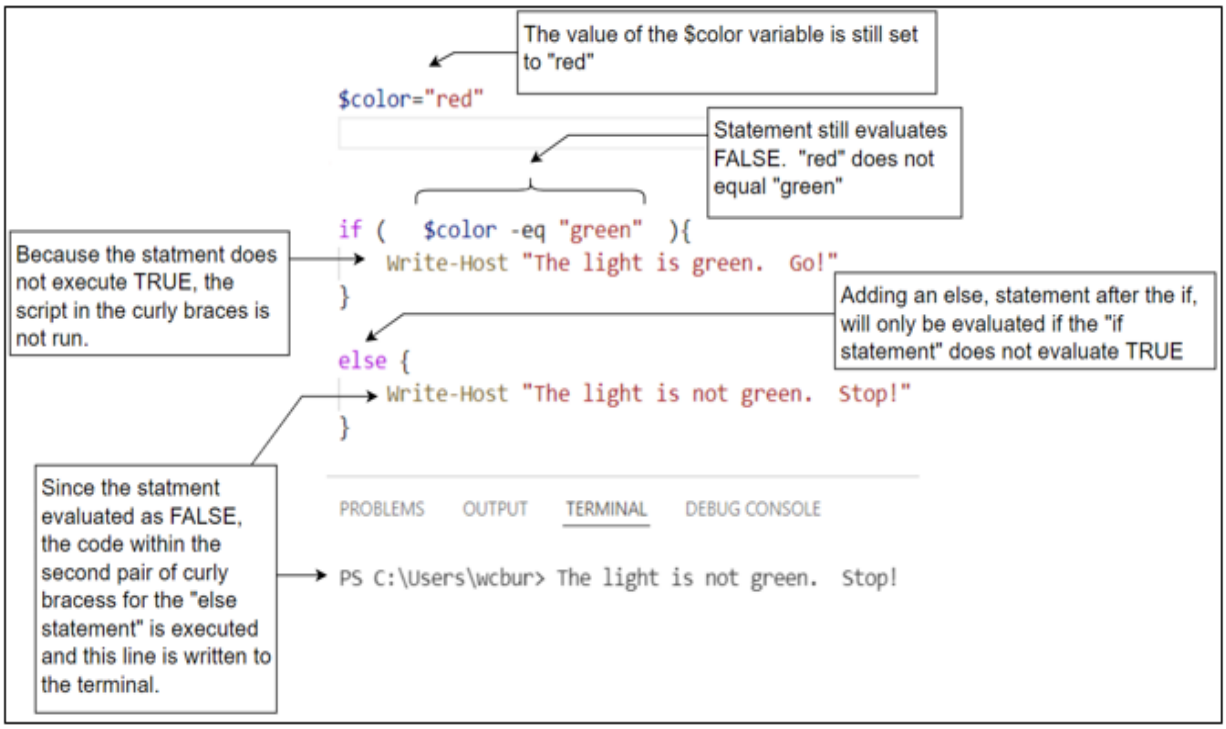
Figure 6.5 If example: False



6.1.2 If / Else

The if statement is a very powerful way to begin implementing conditional logic. But as we can see in the above example, it's often not enough to have a single option. With the script above we can add code to deal with the driving situation if the light is green. But if the light is red our script still doesn't understand how to proceed. We can do this by adding an `ELSE` statement (Figure 6.6).

Figure 6.6 If/Else example



With the addition of an IF/ELSE statement we can now cause the script to behave differently depending upon the conditions we test. This is the foundation of Conditional Logic.

6.2 More Active Directory Methods and Cmdlets

In previous projects we have used Active Directory cmdlets to get and create users, find and unlock locked accounts. Now we are going to use a familiar cmdlet and a new one to copy all the groups one user is a member of to another.

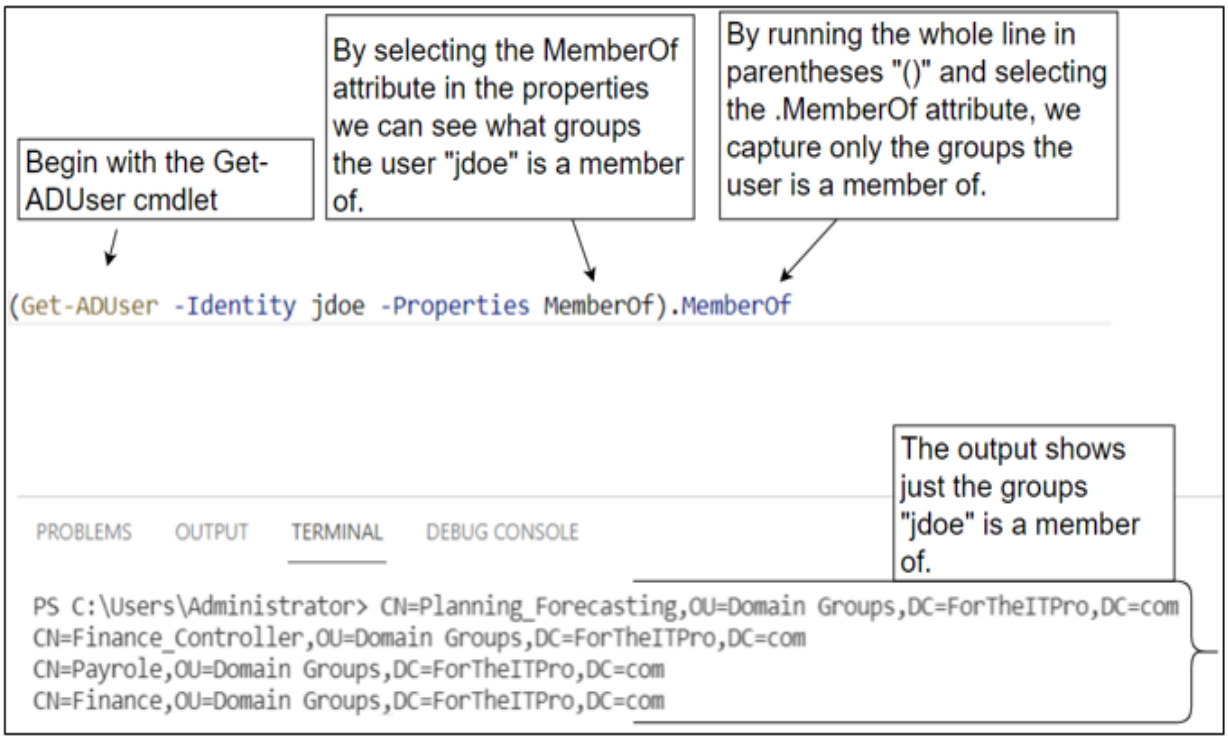
6.2.1 Get-ADUser

We have used the cmdlet `Get-ADUser` before. But Active Directory stores so much information about each user it would be difficult to explain all of the information stored by Active Directory in a single chapter. We have already used this command to find users who are locked out by utilizing the `.lockedout` attribute. Now, we're going to explore the . `MemberOf` attribute.

Memberof

The `Get-ADUser MemberOf` attribute returns distinguished names of all of the groups that this user is a member of (Figure 6.7). However, as a single user has the capability to be in a lot of AD groups (1024 individual groups is the current maximum in Active Directory.), this attribute often does not show all of the groups in the output. PowerShell automatically truncates this output when you run this cmdlet so the output fits on your screen. We'll explore more comprehensive ways to see all of the groups a user is a member of in upcoming chapters.

Figure 6.7 MemberOf example

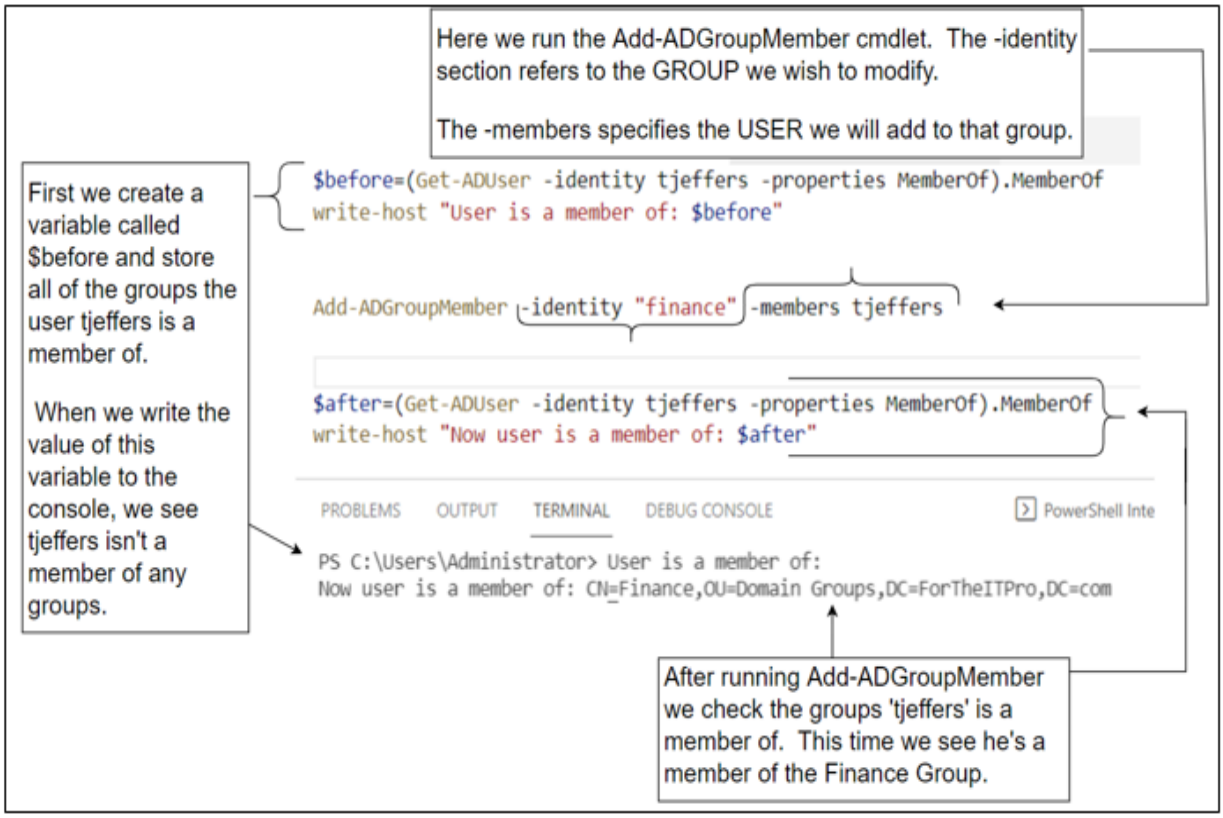


This information is great. We can now copy each of the groups jdoe is a member of to any other user in our domain. But wouldn't it be nice if there was a cmdlet that let us do that instead of having to do it one at a time in Active Directory Users and Groups?

6.2.2 Add-ADGroupMember

Utilizing the Add-ADGroupMember cmdlet we can quickly and easily add new users to existing groups (Figure 6.8).

Figure 6.8 Add-ADGroupMember example

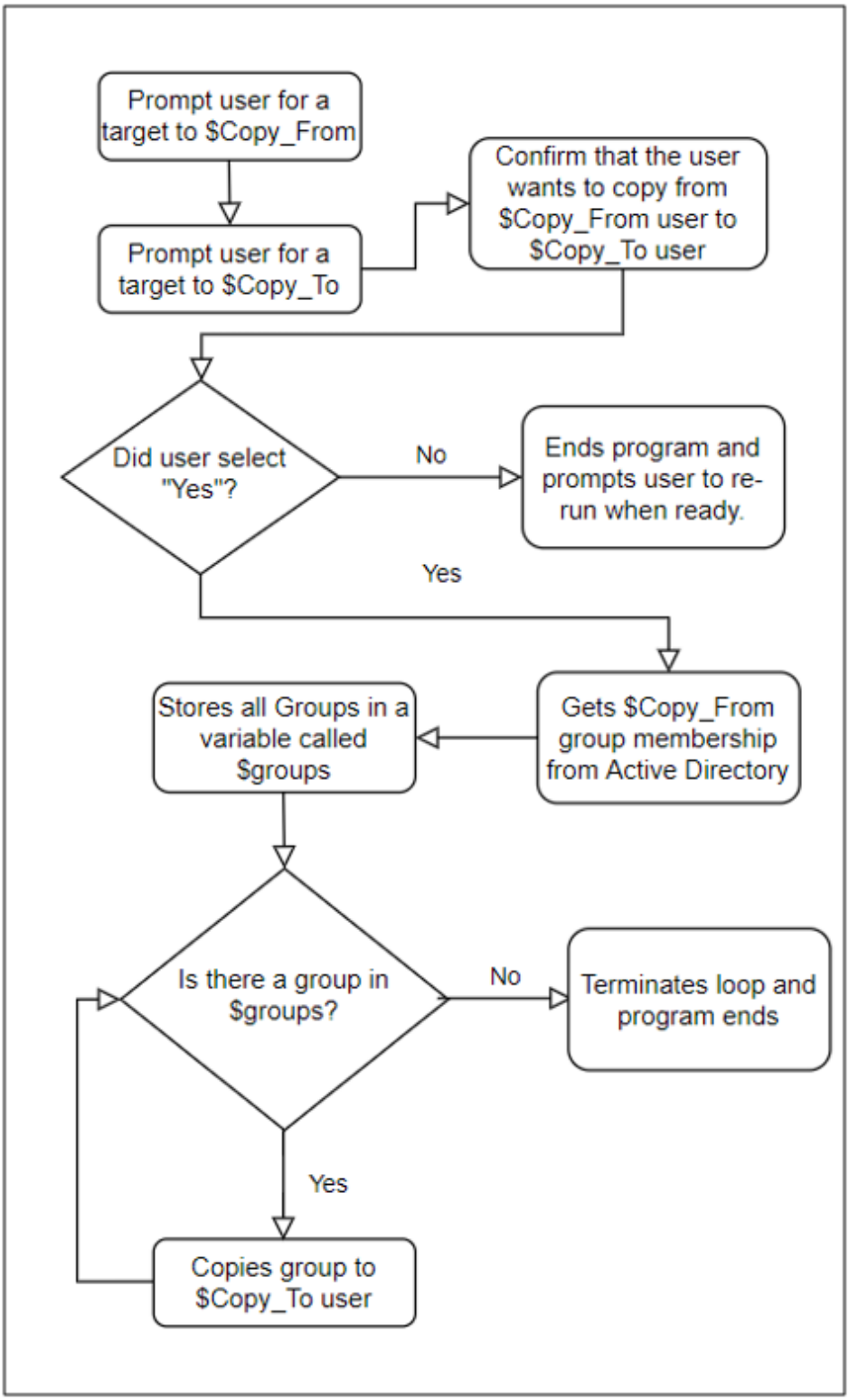


6.3 Putting it all together

Now that we have explored how all of the individual pieces of the Tiny PowerShell project code function. Let's take a moment to examine how we utilize these pieces to make a cohesive script that will make our System Administration life much easier. Pictured below (Figure 6.9) is a flow diagram on how the code will execute.

If you're still new with concepts like loops and if/else statements take a moment here to make sure you understand the branches your code will take to complete the task of copying group membership from one user to another.

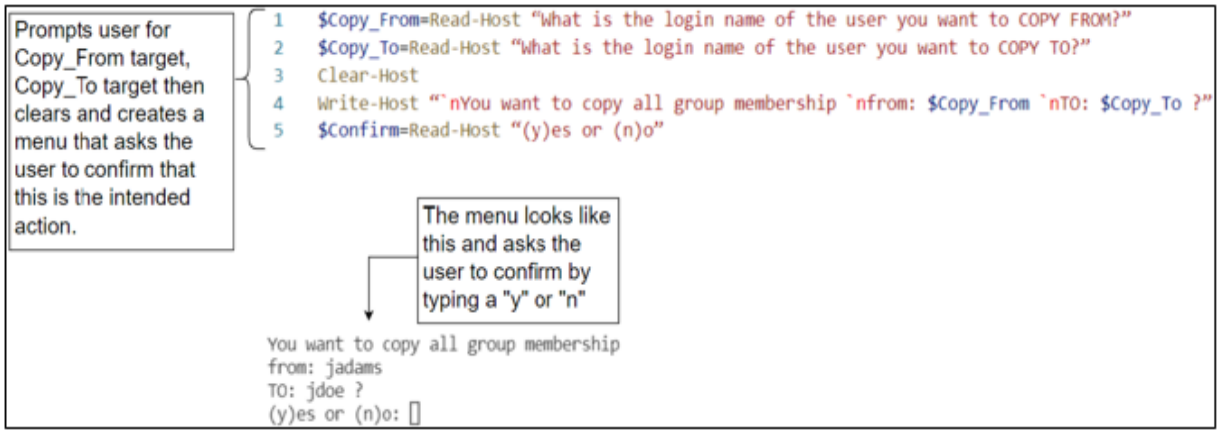
Figure 6.9 Tiny PowerShell project code logic.



6.3.1 Creating a Menu

Utilizing Write-Host and Read-Host statements we're able to put together a Menu that prompts the user to Select two users from Active Directory: one that is already a member of the groups we want and one we will copy those group memberships to (Figure 6.10).

Figure 6.10 creating a menu from Project Code.



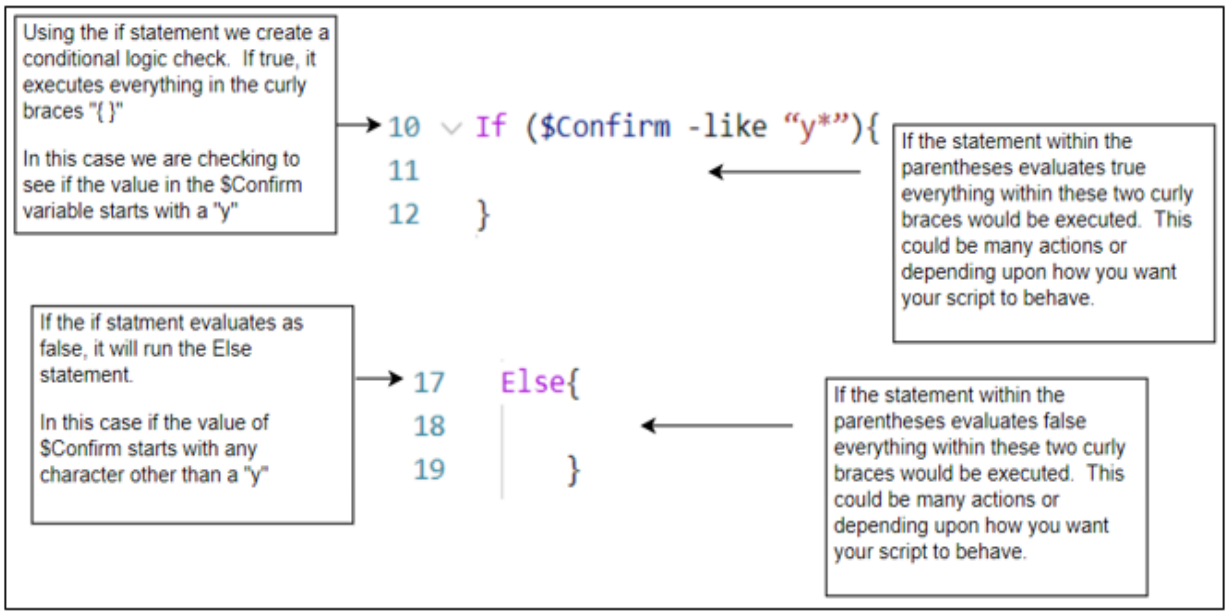
6.3.2 Confirming with the IF/Else statement

Because group membership is sensitive and security relevant, we want to make sure that our user is sure of the changes they wish to make. Once the two users are selected from the menu the user is presented with feedback and asked to confirm with a "y" or a "n".

Using the power of the IF statement we proceed with the remainder of the script only if our user enters a "y" or "yes" (or anything that actually starts with the letter "y").

If the user does not enter something that starts with the letter "y", the script jumps to the Else statement where it prompts the user to re-try the script when ready (Figure 6.11).

Figure 6.11 Utilizing the If/Else statements from Tiny PowerShell project code.



6.3.3 Create a variable containing the groups

MemberOf is an attribute of the objects output by the Get-ADUser cmdlet, if you provide the MemberOf as an argument to the properties parameter when you use Get-ADUser (Figure 6.12). Utilizing this, we are able to store the value of each of the groups that our "$Copy_From" user is a member of and store it in our $groups variable (Figure 6.13).

Figure 6.12 Illustrating how the MemberOf attribute is handled on objects returned from Get-ADUser.

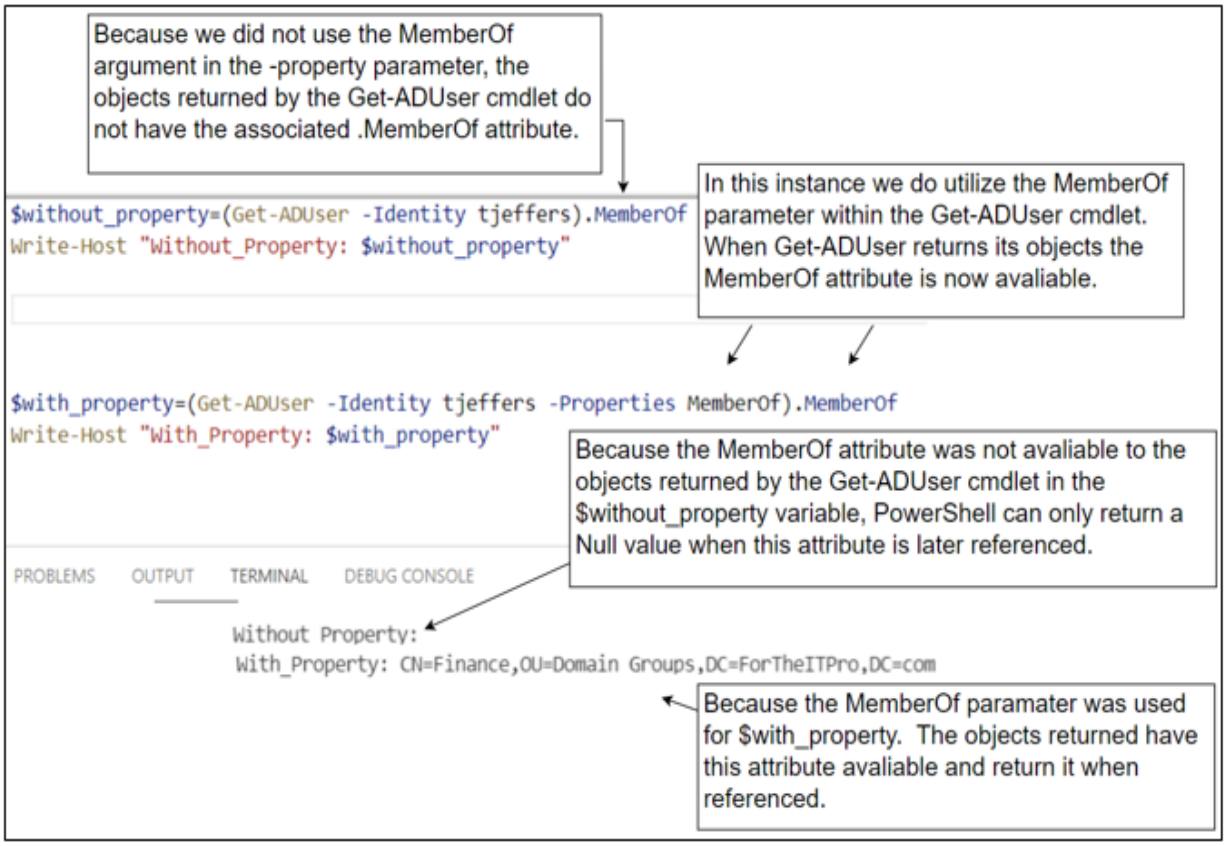
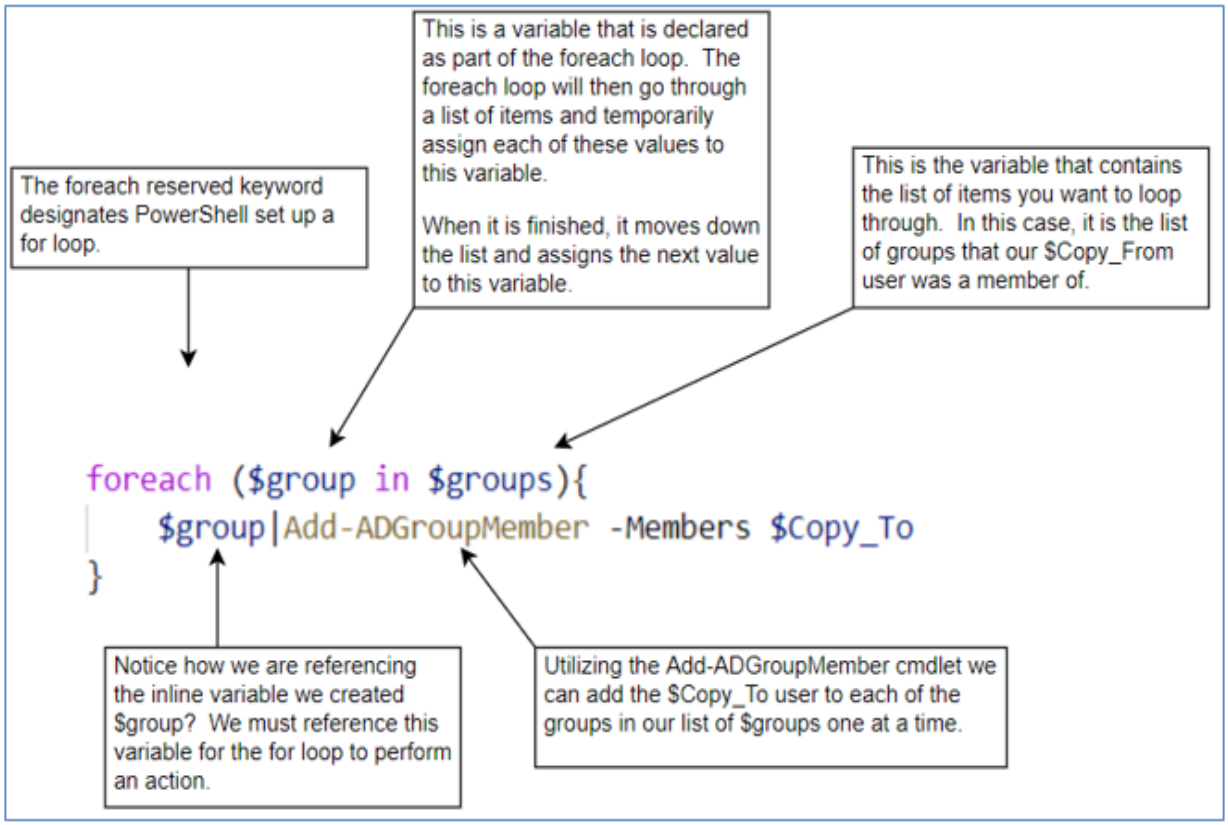


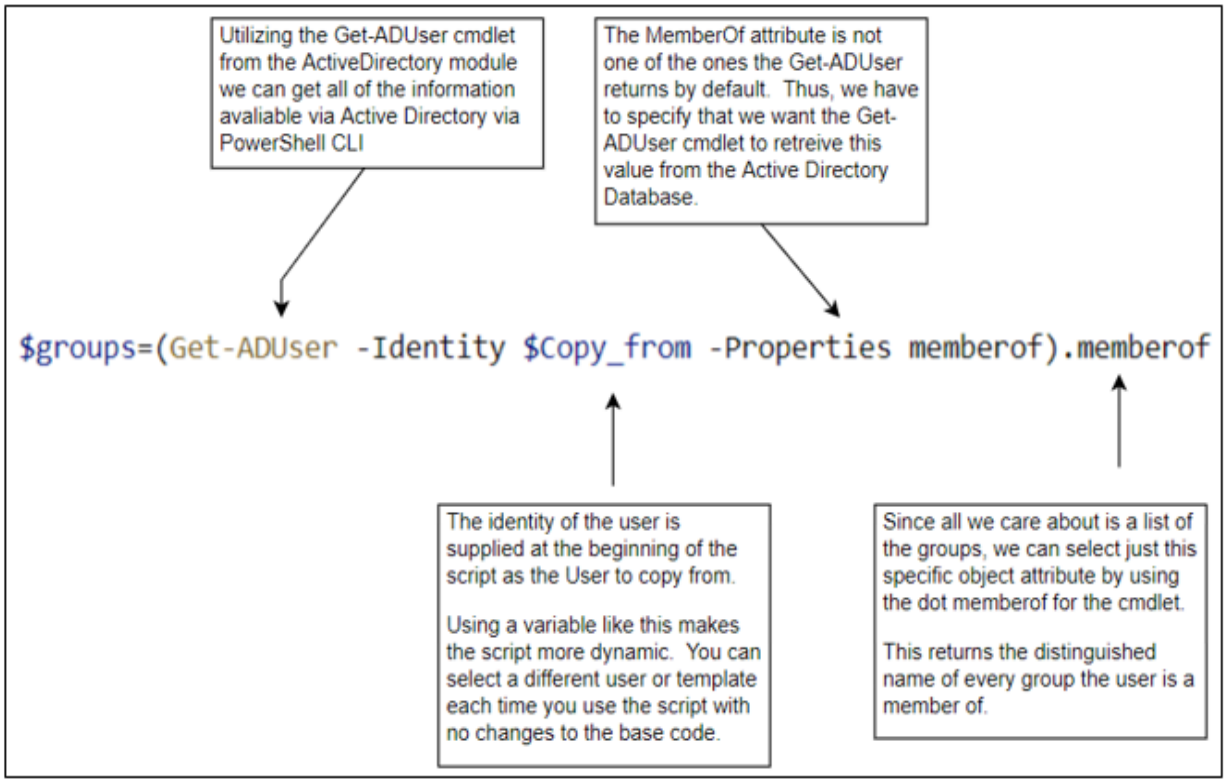
Figure 6.13 Using Get-ADUser to store groups a given user is a member of



6.3.4 Loop through the list

Using a Foreach loop we select each group in our list of groups and then utilize the Add-ADGroupMember to add the user in the "$Copy_To" variable to the group (Figure 6.14).

Figure 6.14 Adding Groups via for loop.



Regardless of which or how many groups the user wishes to copy you can manage your Active Directory Group Membership like a Boss!

6.4 Summary

- PowerShell derives power and flexibility from conditional logic. Utilizing tests and comparisons you can branch your code to treat different results in different ways.
- When utilizing an If statement, your statement will either evaluate true and the actions within the If statement are performed or false, where they are not.
- By utilizing the Else statement, you can perform actions on statements that do not meet the conditional logic in the if-statement.
- The Get-ADUser cmdlet has access to all of the values stored for any user in your Active Directory database. Many of these values are not returned by default but instead are accessed by specifying the name of the attribute in the -properties parameter of the Get-ADUser cmdlet.

- Groups the user is a member of are stored in the Memberof attribute in the Active Directory user database.
- You can add additional group membership to any user by using the Add-GroupMember cmdlet available in the ActiveDirectory module.
- Looping through a list of these groups can quickly and easily add a targeted user to any number of Active Directory Groups.

7 One-click Exchange account fix

This chapter covers:

- How to build a function that we can utilize multiple times in our script.
- A new type of loop: the Do-Until loop perfect for status updates.
- Built in Exchange functions that lets us create, remove, backup and import user mailboxes.

At the time of writing this, email has been and is the king of internal communication. With the rise of social media and the ever-present text and phone communication options email has lost some of its ride or die status; yet it's clear that this form of communication is not going anywhere soon. As a system administrator it's likely you'll be tasked with some common Exchange related tasks.

Currently there are three main types of Exchange solutions offered to your typical enterprise.

- On-Prem
- As a Service
- Hybrid

One thing they all have in common? You, as a system administrator, are going to be interfacing with them. Even with your email as a service and hybrid solutions utilizing Office 365, management of your email systems is up to you.

Most system administration of Exchange services is done either through ECP (Exchange Control Panel) or the EAC (Exchange Admin Center), but there is a third option: PowerShell.

Both EAC and ECP are Web-Frontend GUIs, but you can fully control every aspect of your Exchange system through the Exchange Management Shell built on Windows PowerShell. This means that with the use of PowerShell scripts you can administer your entire Exchange environment faster, easier

and more efficiently. Typical administrators using the GUI will be forced to modify mailboxes one at a time. Utilizing the skills you've already learned such as variable arrays, conditional logic and loops you can administer hundreds or thousands with a script and let the computer do the work.

One of the most common fixes for broken exchange mailboxes is simply to disable and re-enable the mailbox. This essentially re-creates the user's mailbox and fixes all kinds of issues from mailbox corruption to database integrity issues. It is not, however, without its downsides. Primary among these is the fact that when this is done, the user's old email is no longer associated with the new mailbox.

If your email users are anything like mine, they use their email boxes as their own personal time machines going back and pulling that one critical email out of the message ether from fifteen years ago. Thus, the thought of losing their stores of historical emails is as terrifying as the latest Friday the 13th movie; and much more real.

The easiest way to resolve this is to back up the user's .pst file (Offline Outlook Data File). This holds the user's emails, calendar items and other things. With the .pst file saved we can safely re-create the user's mailbox and re-import the .pst and restore all of the historical data.

The typical issue with this is that backing up a .pst file, especially for a large mailbox, is a long process. Relying on a user to back up their own .pst file can work in some situations; but destroying a user's email box and relying on them to back up their own messages can lead to nightmare scenarios where the data is gone for good and you're dealing with an enraged user bent on making you feel their pain.

This usually translates to:

- System admin goes to user's computer (or remotes in)
- Starts backup process of .pst file
- Waits several minutes to several hours for the process to finish
- Disables the user's mailbox via EAC
- Enables the user's mailbox via EAC
- Re-imports user's .pst file

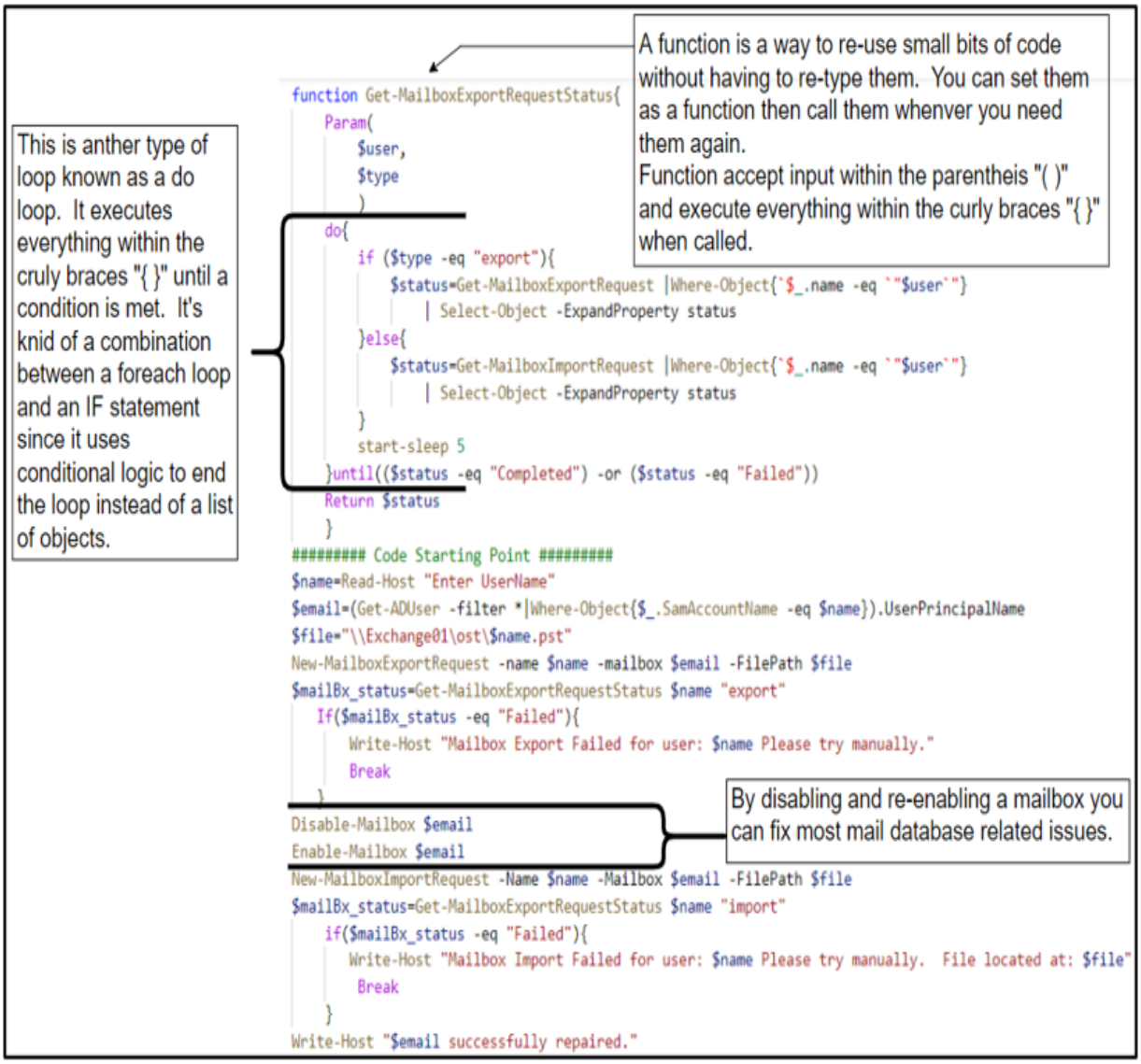
This can be fine, however, time consuming for individual users. But if you have a significant corruption of a database affecting many users, this process could take several days before you're able to restore your users. PowerShell to the rescue!

Utilizing this script (Figure 7.1), you'll be able to:

- Type in a user's username.
- Automatically begin the backup of the .pst file
- Within 5 seconds of it being complete we disable the mailbox
- The mailbox is then re-enabled
- Automatically begin the re-import of the .pst file.

No more wasted time! While the script is running the system admin is free to address the many other issues that come into play with a major Exchange issue, including communication of the outage and release of status reports, and he/she is able to give status updates to upper management without ever having the ETA affected.

Figure 7.1 Tiny PowerShell project Code



Note

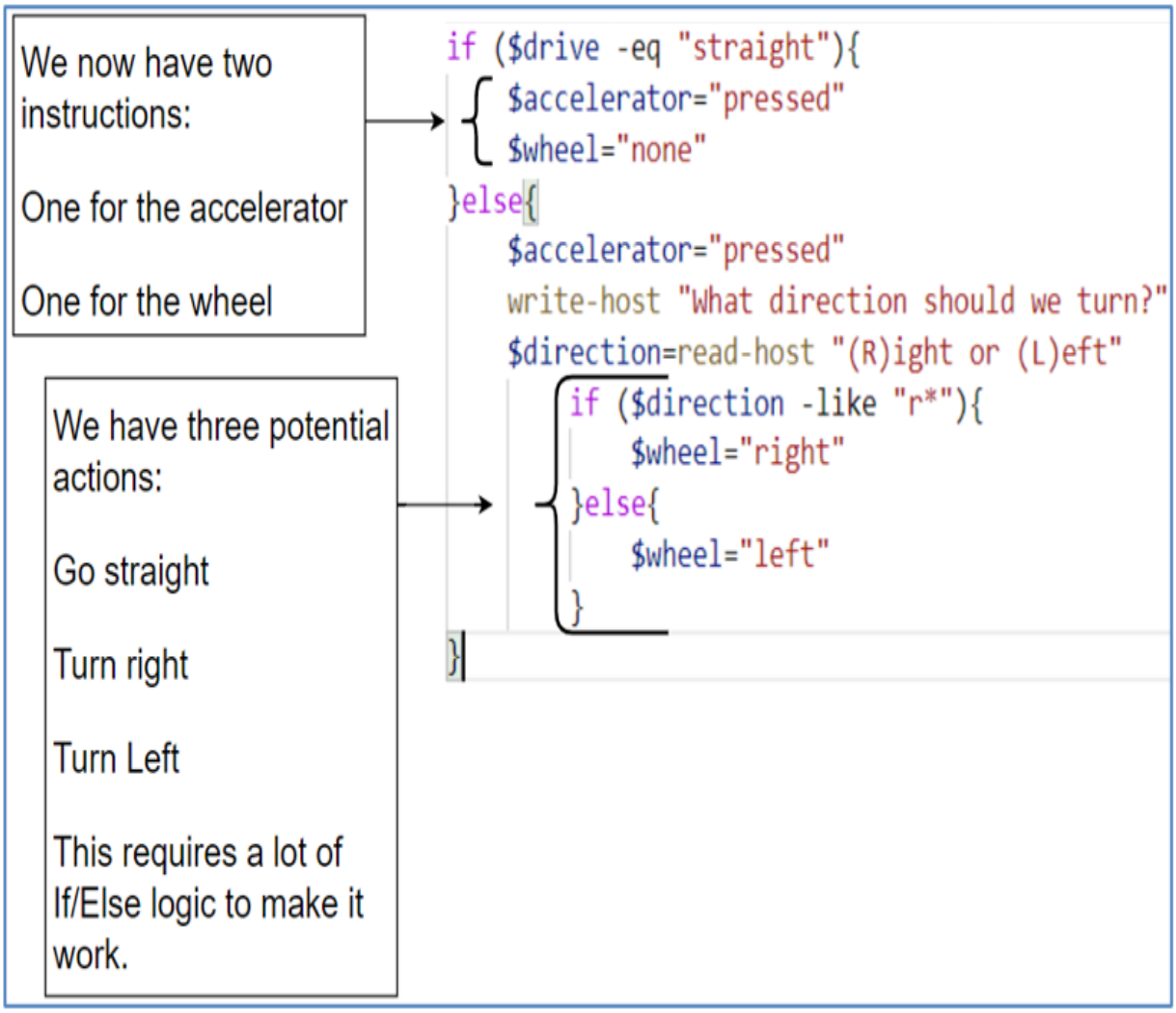
In future chapters of this book, we will discuss how to connect to remote PSSessions and Import modules, aliases and functions into our current, working PSSession. This will allow us the flexibility of running the Exchange cmdlets on any PC in our domain. At this point in the book, however, unless you are already familiar with PSSessions and Importing-PSSessions, it is recommended that you perform this script and related cmdlets while remoted into an Exchange server on your domain.

7.1 Functions

A function is a grouping of code that can be called, much like a cmdlet to execute specific tasks. The less code you need to write the less chances you'll have a bug. Frequently, as you progress in your programs you find yourself needing to make minor changes to your code. Having code that you utilize over and over encapsulated in a function makes it so you only have to make modifications to the code in one single place.

In the last chapter we looked at how we might program a car to drive itself. But the process of driving is fairly complicated. You must press the accelerator, or the brake. You must steer and utilize turn signals. Let's take a quick look at how we might instruct a car to drive (Figure 7.2).

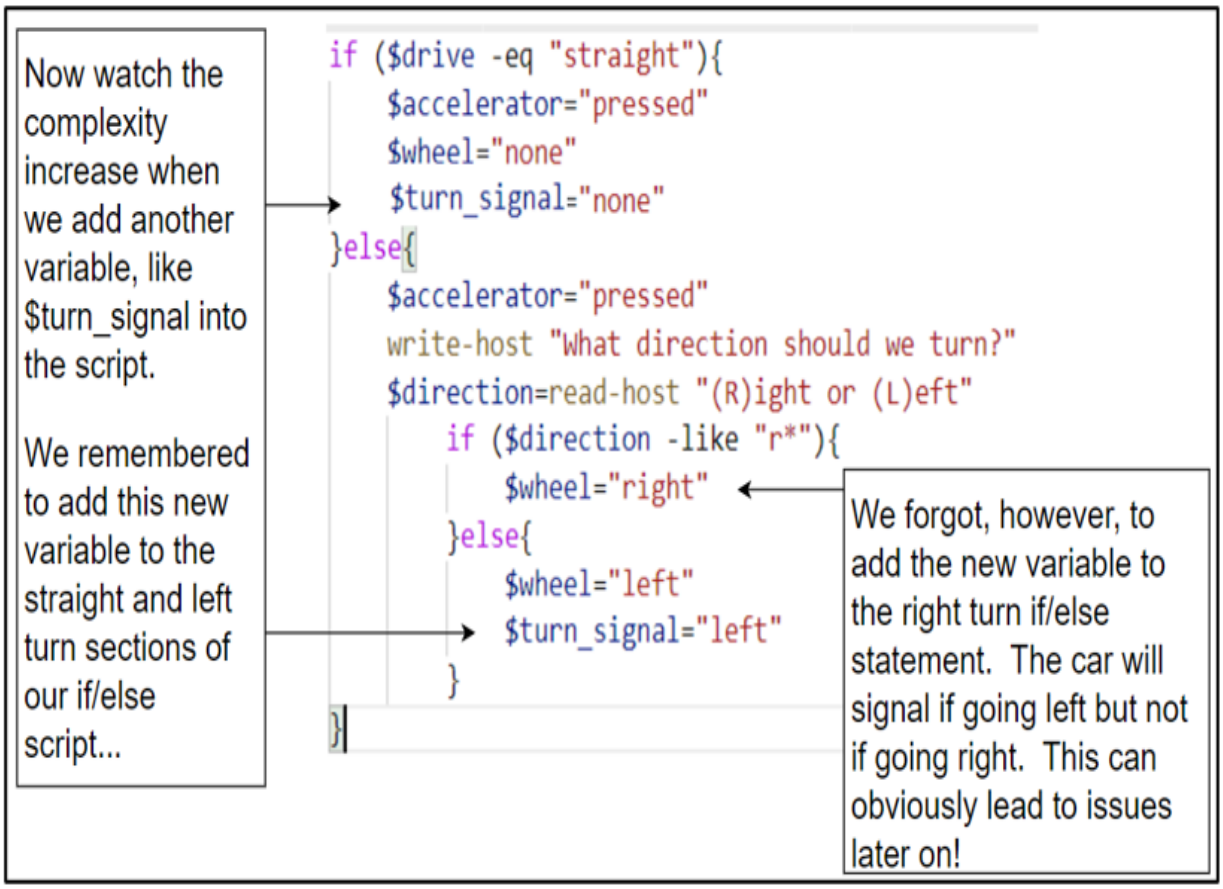
Figure 7.2 Turning a car, the hard way



In order to instruct the car to press the accelerator and turn we must create two new variables. $accelerator and $wheel. But to instruct the car to go straight, left, or right we need a total of four if/else statements. In addition to this we need to keep track of the variables $wheel in three different places. If you later decide that the name $wheel is not specific enough (as you might be referring to the wheels of the car later in the script) and want to change the variable to $steering_wheel, you must now change it in all three places or potentially introduce a bug to your script.

Let's look to see what it takes to add a turn-signal to the script (Figure 7.3).

Figure 7.3 Why functions can save us time and hassle

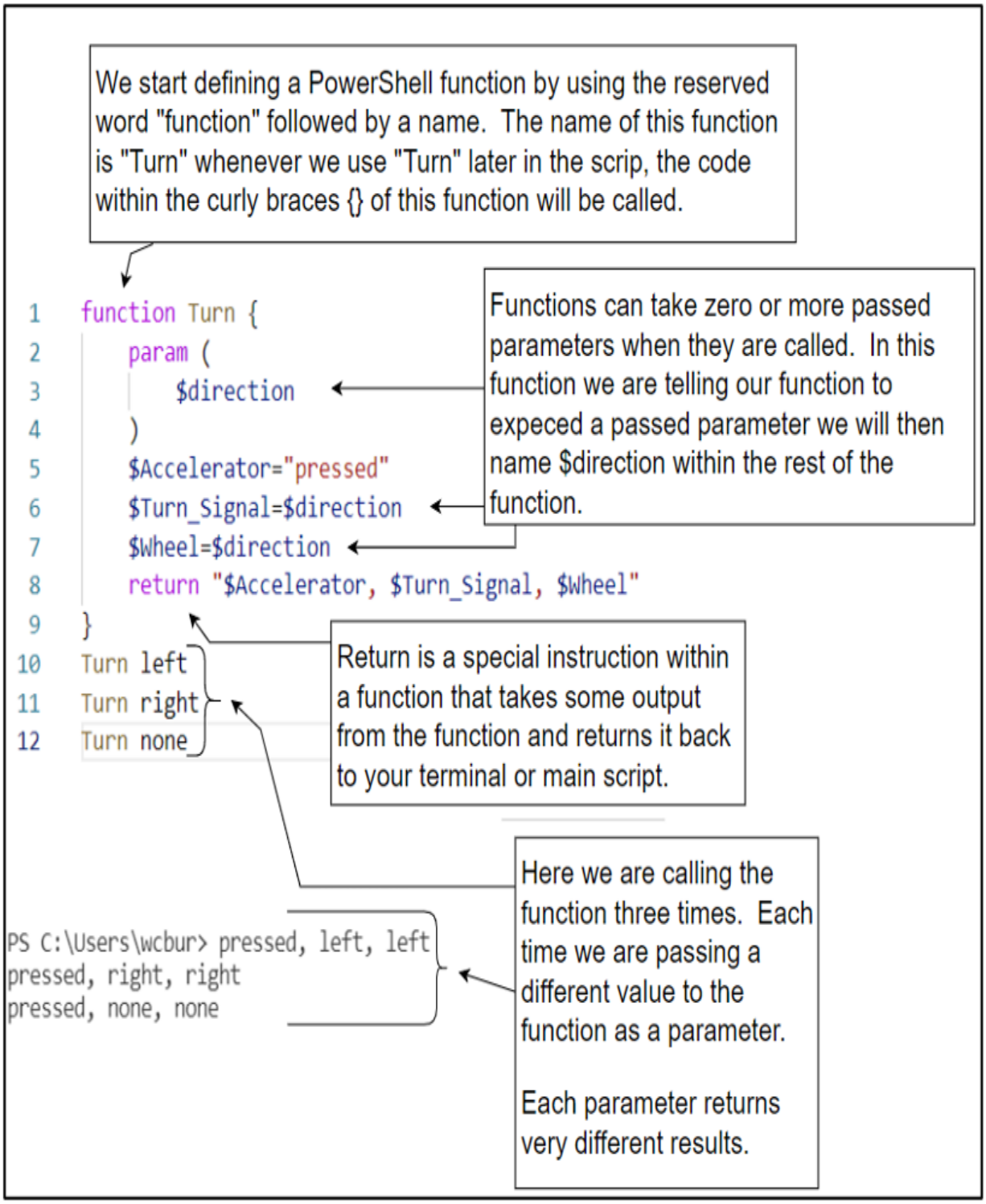


Wouldn't it be easier to simply tell the car:

- Turn Right
- Turn Left
- Turn None

We can use a function called "Turn," shown in Figure 7.4, to do exactly this!

Figure 7.4 Function example



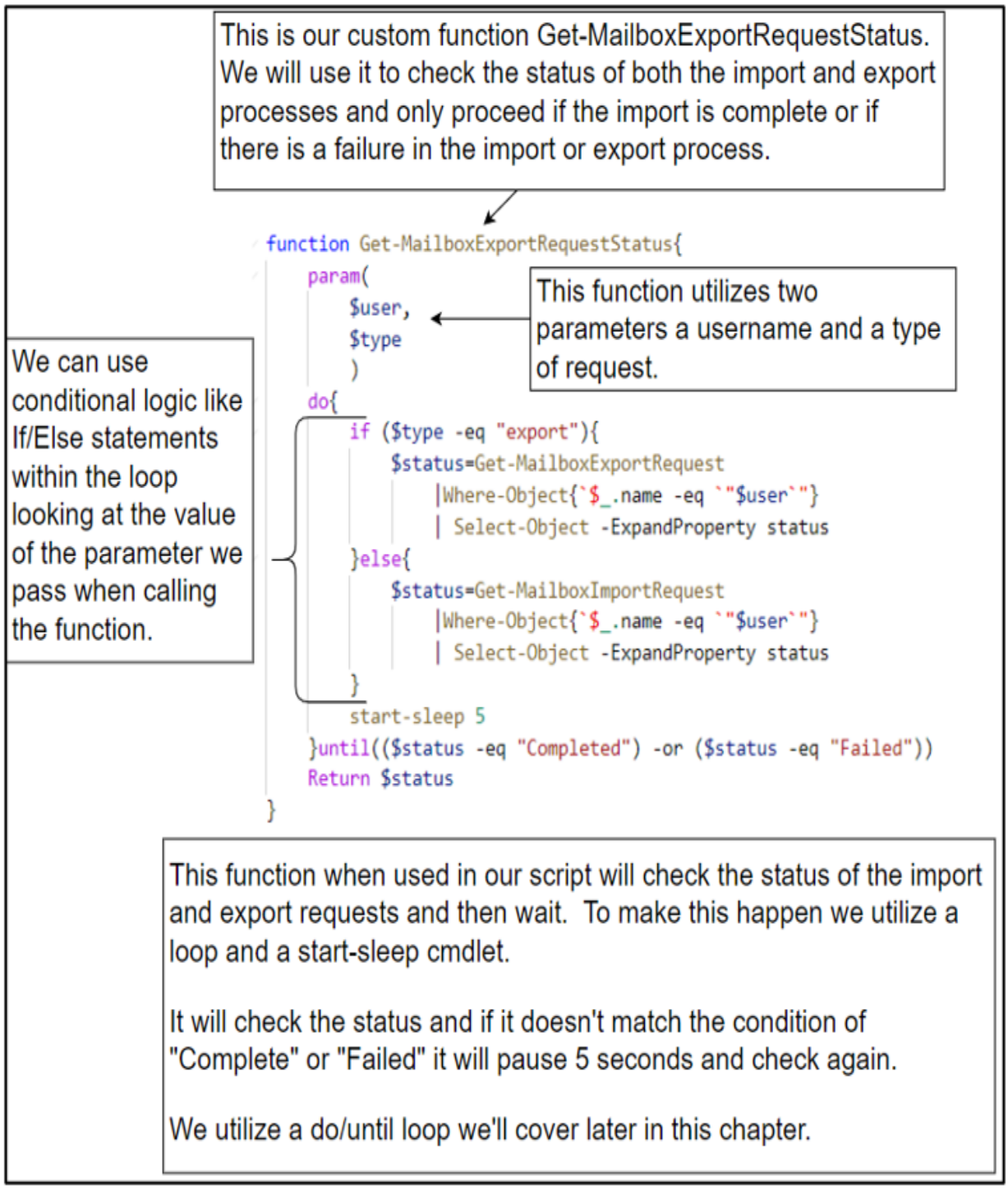
As you can see, in the example above we actually called the function Turn, three times. If we wanted to do the same thing with the If/Else statement we

first used we'd have to type all of those If/Else statements three times supplying a different value for drive each time.

We were able to accomplish exaclty as much functionality with 9 lines of code that we would with 48 utilizing the If/Else statements shown above.

In our Tiny PowerShell project, we will be setting up a function that checks the status of a mailbox import and export of the user's mailbox. Only when it's fully backed up/restored will the code then proceed. This function prevents the script from disabling the mailbox before its contents were backed up. I'll preview the snipit of the code here, in Figure 7.5. Don't worry if you don't fully understand what it's doing at this point; we'll cover the other things being accomplished later in this chapter.

Figure 7.5 Tiny PowerShell get-status function



7.2 More Looping

We have seen loops a in a number of scripts now. Except for Chapter 4 Bugs, where the loop wasn't explained, each time we've used them they have been

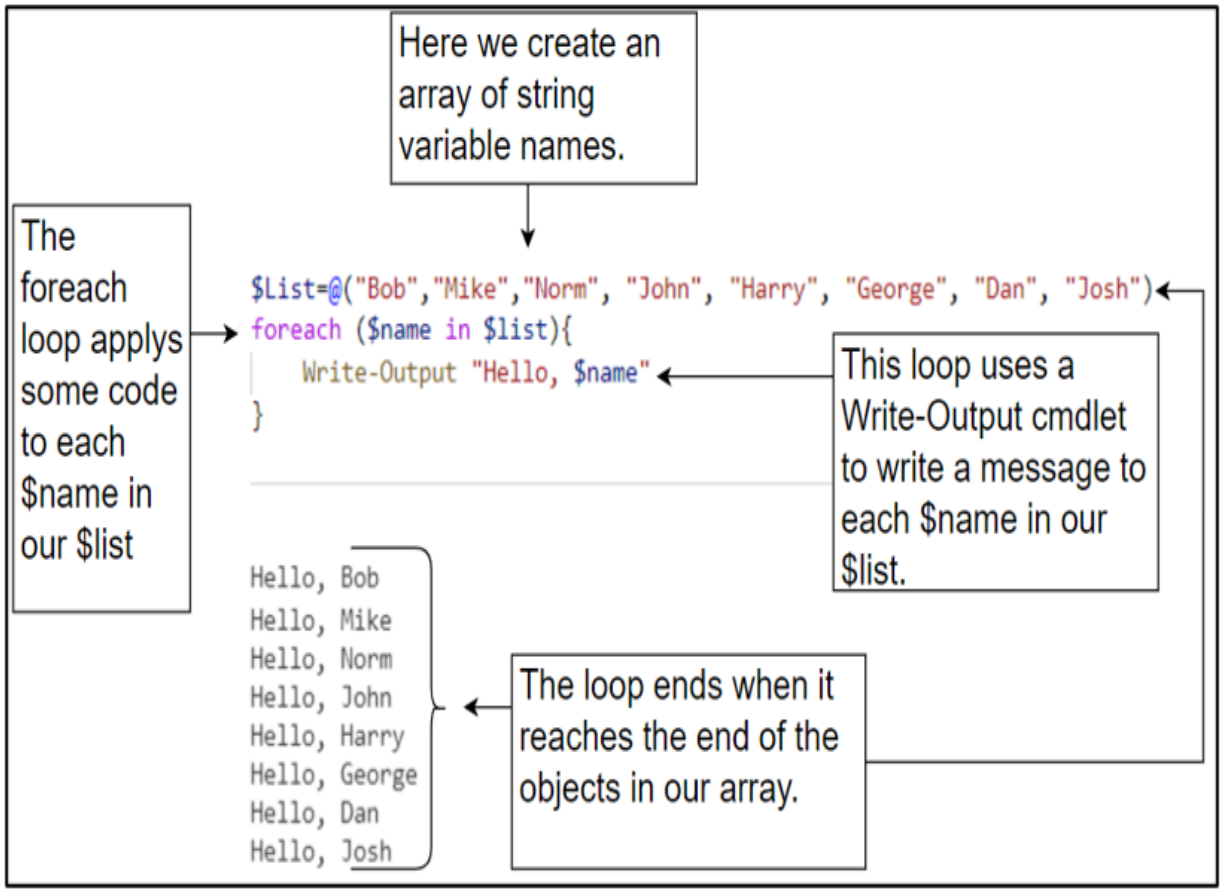
the *foreach loop*. But there are many types of loops.

7.2.1 Foreach Loop

Let's quickly review the steps in a foreach loop (Figure 7.6). They are:

- We have a number of items in some kind of list or array.
- We have a specified amount of code we wish to apply to each of them
- We loop through the whole list and apply the code to each item.
- When there are no more items in our list or array the loop exits.

Figure 7.6 Foreach loop review



In our script for this Tiny Project, we utilize the ***Do-Until Loop***. The **Do-Until Loop** runs in a loop until an exit condition is met. If we have a situation where we want to exit a loop when an error is detected, we would

have to add an ***IF*** statement and a ***Break*** command to exit the loop if an error was detected. The **Do-Until Loop** has this built in. It's kind of a combination of a loop and an **If** statement.

7.2.2 Do-Until Loop

While the foreach loop runs on every item in a list or array, the Do-Until loop will check after each iteration of the loop to see if it should exit or continue. The Do-Until loop works like this:

- Runs the Code on the first item in the loop
- Checks the Exit condition
    - If the Exit condition is met it exits the loop.
    - Otherwise, it continues to the next item in the loop
- This process continues until the exit condition is met.

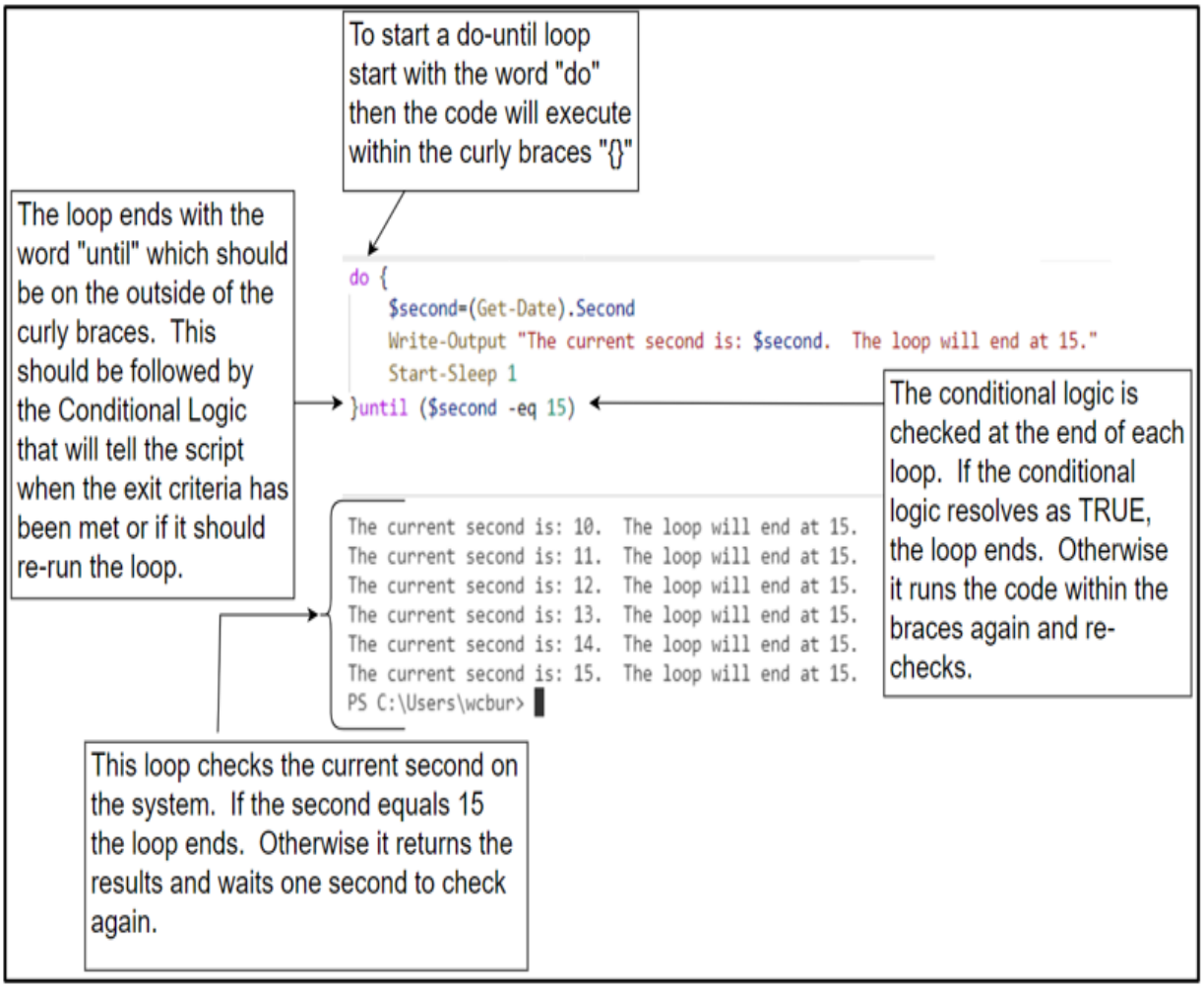
In the example below (Figure 7.7) we'll get the system time, just looking at the current second, then sleep for one second. If the second equals 15 the loop will end. This means that if we run this loop at the top of the minute, the loop will run 15 times checking the first second then the second and the third until it gets to the 15th second. If start the loop on the 16th second it will run 59 times. After each loop the Do-Until loop will check to see if the exit condition is met and if it hasn't it will run the loop again.

Get-Date

To accomplish this, we'll need a way for PowerShell to get the current second from the system time. Fortunately, there's a cmdlet already build for this functionality called Get-Date. The Get-Date cmdlet gets a DateTime object from your system and is capable of formatting it in many ways.

By default, Get-Date returns the Day of the Week, Month, Date, Year, Hour, Minute, and Second. But we can easily isolate any of these elements, to retrieve only the current second, we use the dot notation to request only the seconds parameter from the returned DateTime object.

Figure 7.7 Do-Until loop example

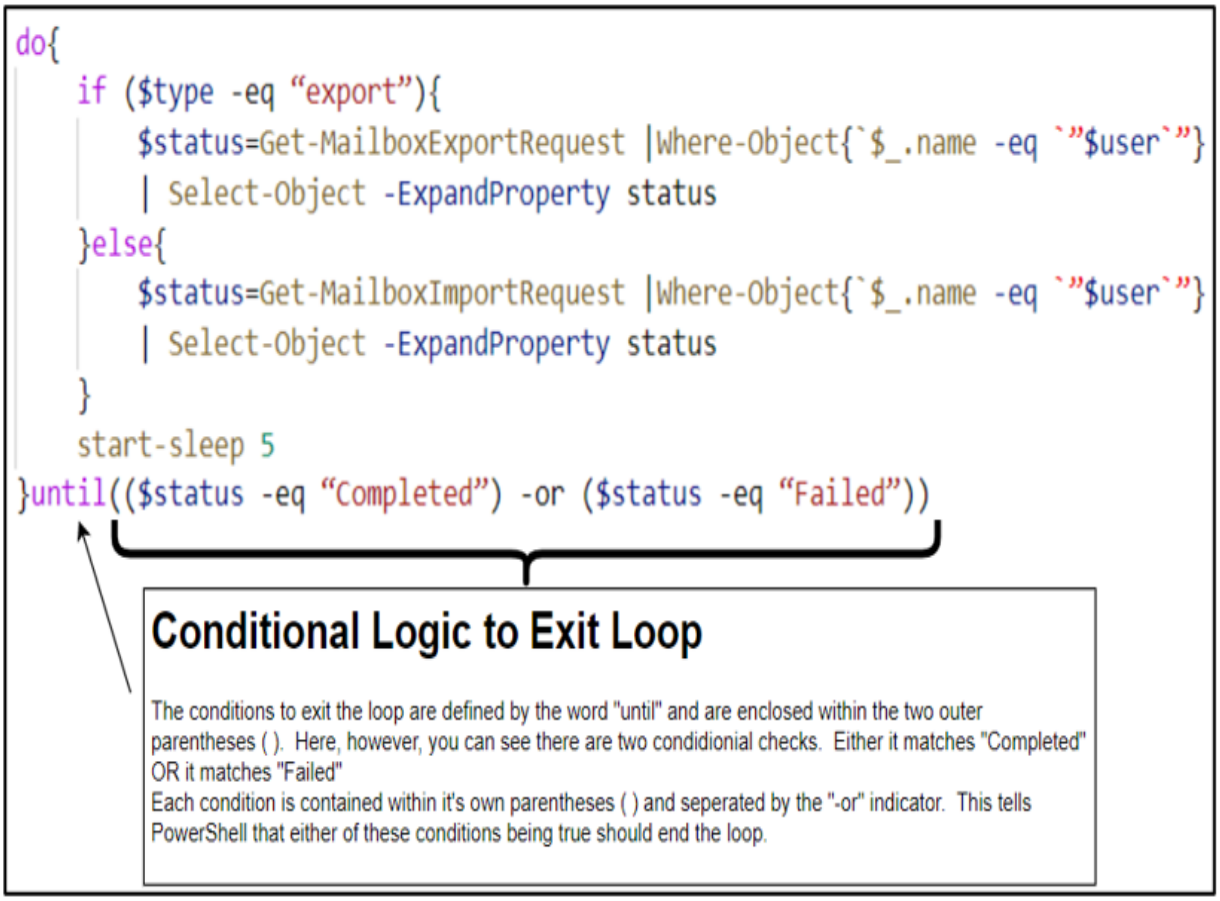


The "Or" Conditional Logic

There are times when using conditional logic where we need to check for multiple conditions. The logic we are trying to apply might be useful if, some, none or all of the conditions are true.

Let's look at the conditional logic we are using to end our do loop, shown in Figure 7.8.

Figure 7.8 Or comparison operator example



Here there are several different types of statuses that could be returned:

- Queued
- InProcess
- Completed
- Failed

We don’t want to exit our do loop until the process of backing up the .pst has finished. But we don’t want to proceed with disabling the mailbox if the .pst backup has failed. So, we really care about only two of the four possible results of the $status variable (Completed and Failed). We can set either of these conditions as a way out of our **do-until** loop with the inclusion of the “-or” statement in our Conditional Logic.

Now that we understand the structure of the loop let’s look at what we can use the loops to perform. In our Tiny PowerShell project code, we are using the **do-until** loop to wait until the status of the ImportMailbox or

ExportMailbox command is complete. But how do we check the status to see if the condition is met? We can use PowerShell to query the Exchange server and check.

We could write multiple functions that do very specific things. Or we can try to make a more dynamic and universal function that allows us to perform different functions depending upon the types of parameters we pass it when calling the function. Using these parameters and conditional logic we can construct a very specific Exchange server command.

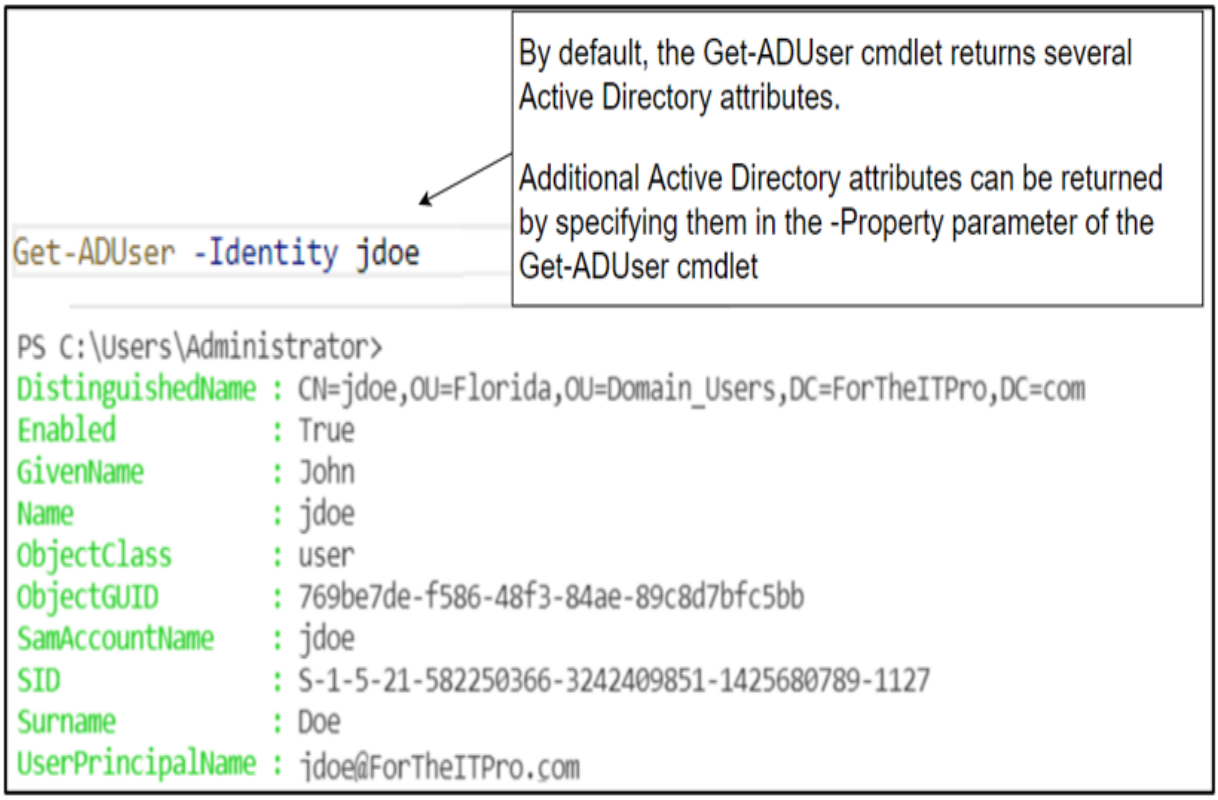
7.3 Select-Object

Select-Object is a cmdlet that allows us to select an object or part of an object and return that value. PowerShell and scripting in PowerShell is all about learning how to manage the objects you care about. Select-Object can be used to select specific number, types and parts of objects all of which make our lives as PowerShell scripters easier.

Select-Object -Property

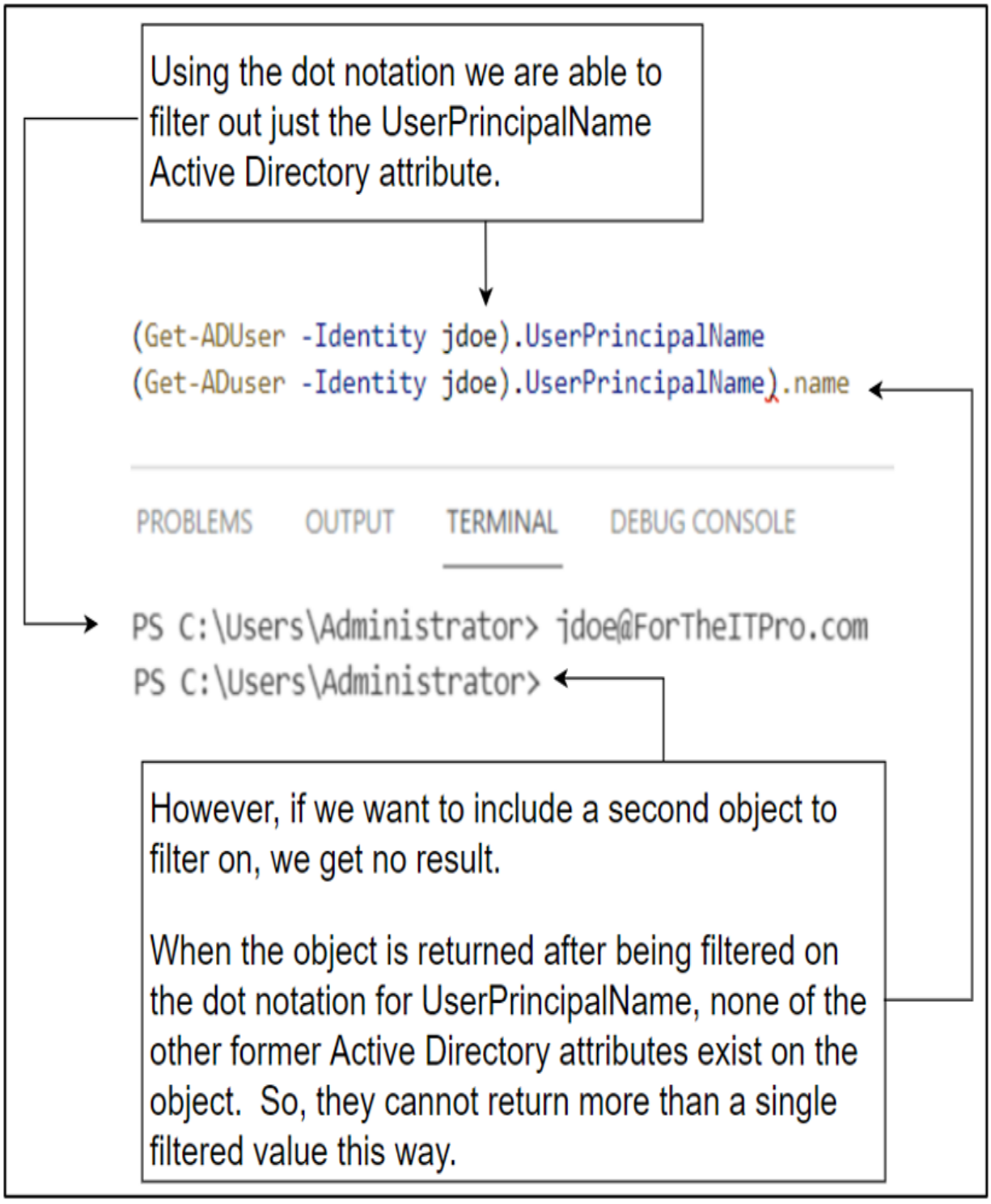
There are many times where the results of a command are simply more than we care about. If, for example we were only interested in a complete list of email addresses in our Active Directory Database, running a Get-ADUser cmdlet and having the user's Name, SamAccountName, DistinguishedName and all of the other values that are returned by default from that cmdlet (Figure 7.9) would be something we would need to filter out.

Figure 7.9 Default output of Get-ADUser



We already have seen this type of filtering using the dot notation in PowerShell. But this is limited by the fact that we can only select one parameter at a time utilizing the dot notation on a cmdlet. If we wanted to create a list that included a user's email address and full name, for example, we could not use the dot notation (Figure 7.10). Because when we use the dot notation the object returned no longer contains any of the other Active Directory attributes.

Figure 7.10 Trying to filter for more than one attribute with a dot notation.



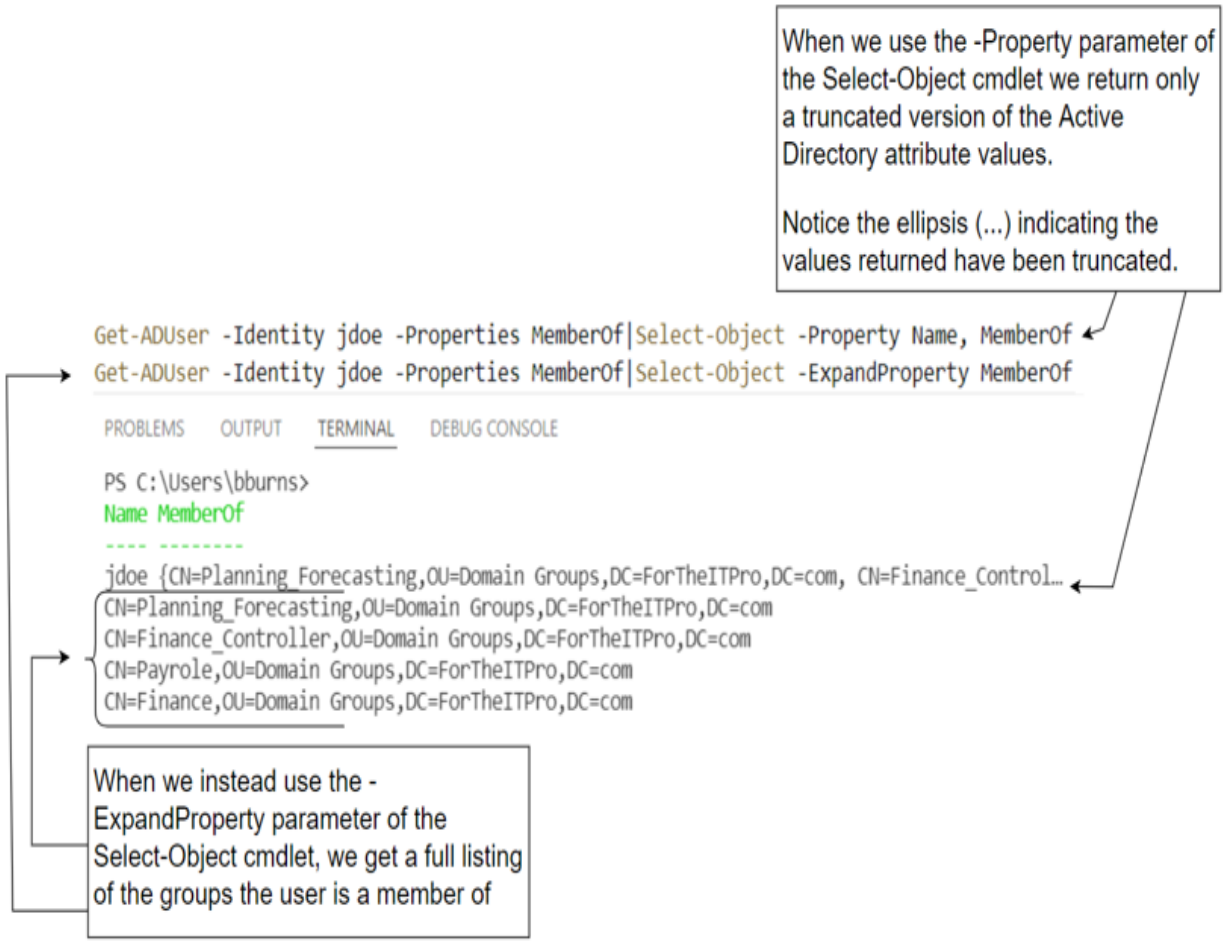
Utilizing Select-Object, however we can select multiple Active Directory attributes with the -Property parameter (Figure 7.11). Here we can select for one or more attributes on the object with ease.

Figure 7.11 Using Select-Object to select more than one property from a returned cmdlet.

Select-Object -ExpandProperty

Some object properties are a collection of values or nested objects. When you try to display them, they may not display properly. We can utilize our MemberOf Active Directory attribute as an example. The MemberOf parameter in the Get-ADUser cmdlet returns the groups the user is a member of. But these could be many, many groups. The -property parameter of the Select-Object cmdlet will not return them all. Instead, it will truncate the output to indicate that there are more objects that are not being displayed. Utilizing the -ExpandProperty parameter, however, will return all of these values (Figure 7.12).

Figure 7.12 -ExpandProperty example with Get-ADUser MemberOf attribute.

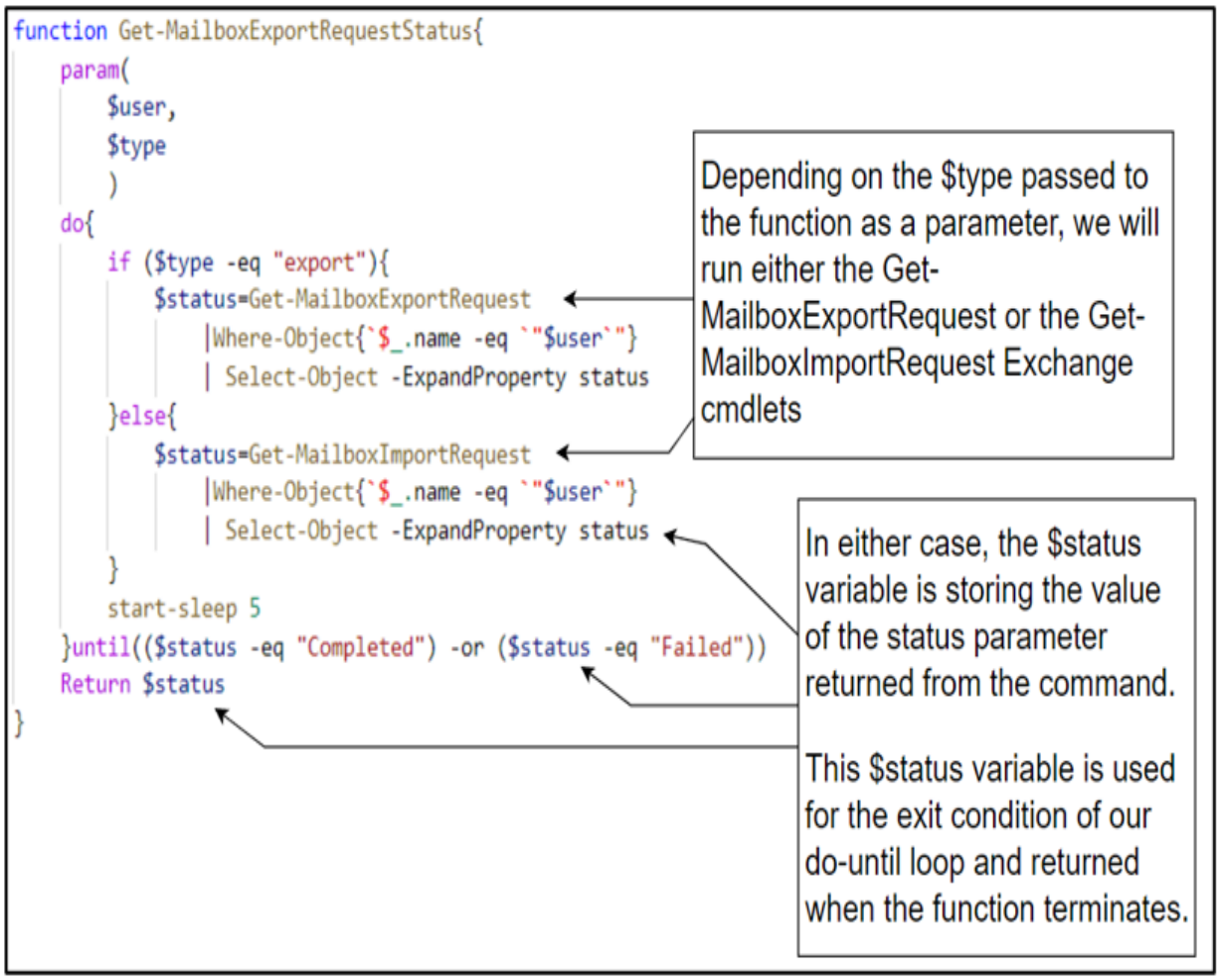


The drawback to utilizing the -ExpandProperty parameter of the Select-Object cmdlet is that like the dot notation, we can only expand a single property. So why, you might be wondering, would we use the Select-Object instead of the dot notation if we're limited to a single property anyway. Both simply output the value of the property in a string (plain text), which one you chose to use depends largely upon your preference and style selection.

Using this in our script

Utilizing our Function, we are going to run either the Get-MailboxImportRequest or the Get-MailboxExportRequest cmdlets, depending on the parameter we pass to our function. We then utilize the Select-Object and -ExpandProperty parameter to pull back just the status parameter for the Exchange cmdlet. Let's look at our Tiny PowerShell code, shown in Figure 7.13.

Figure 7.13 Status captured by Tiny PowerShell code function.



Notice that there are a few strange characters in the Where-Object part of our Get-MailboxExportRequestStatus function? There are tiny backticks in front of the $ and both quotation marks. These are known as escape characters. See the sidebar below for more information on escape characters if you're unfamiliar with how they work.

Escape Characters

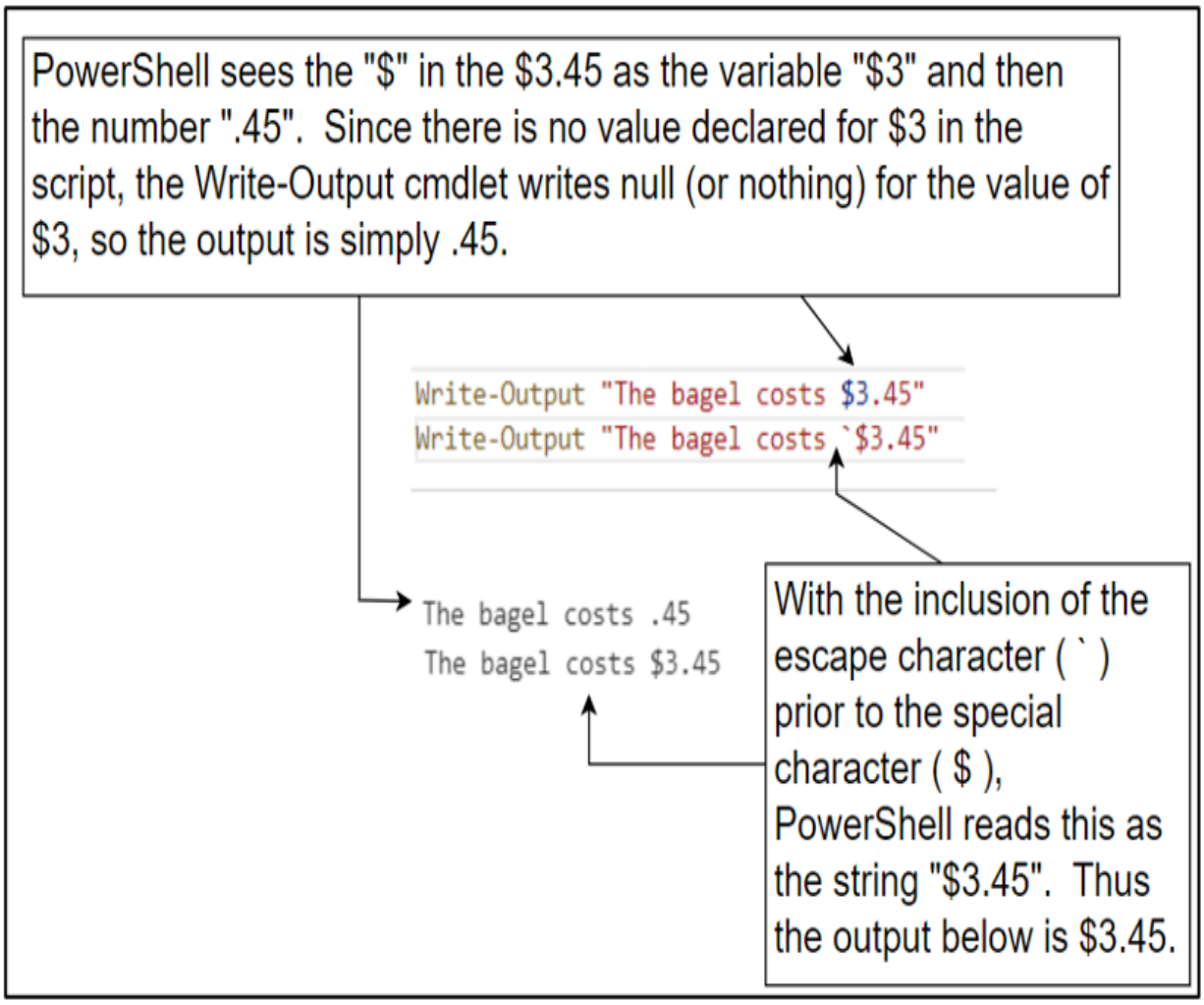
Because specific special characters have unique meaning in the PowerShell language there are times we will need to use an escape character. The escape character is simply a way of telling PowerShell to treat the character as a normal character and not a special character (Figure 7.14). PowerShell's escape character is the backtick (`) character. This may look like a single

quote or an apostrophe. But it is the character just to the left of the number "1" key on a standard QWERTY keyboard.

Be aware, that sometimes the backtick is part of a special character.

`n, for example, which we have used before to indicate to PowerShell we want to start a new line within a string. In these cases, the backtick doesn't tell PowerShell to treat the character like a normal character but instead performs a special function.

Figure 7.14 Escape Character Example



7.4 Exchange Specific Cmdlets

Microsoft Exchange is built on a PowerShell backend. There are dozens of Exchange Related cmdlets that will help you administer your email services. However, because this is a rather specific application of the PowerShell language; these cmdlets are not standard in the cmdlets bundled with your typical PowerShell 7 download.

Much like with Active Directory, working natively on a Domain Controller, the Exchange cmdlets are most accessible on your Exchange Server. Therefore, until we cover how to access a federated service, like exchange, on your desktop, it's probably best that the majority of your Exchange Scripts be run on the Exchange Server or Exchange Admin station.

We are going to be focusing on a handful of Exchange related cmdlets:

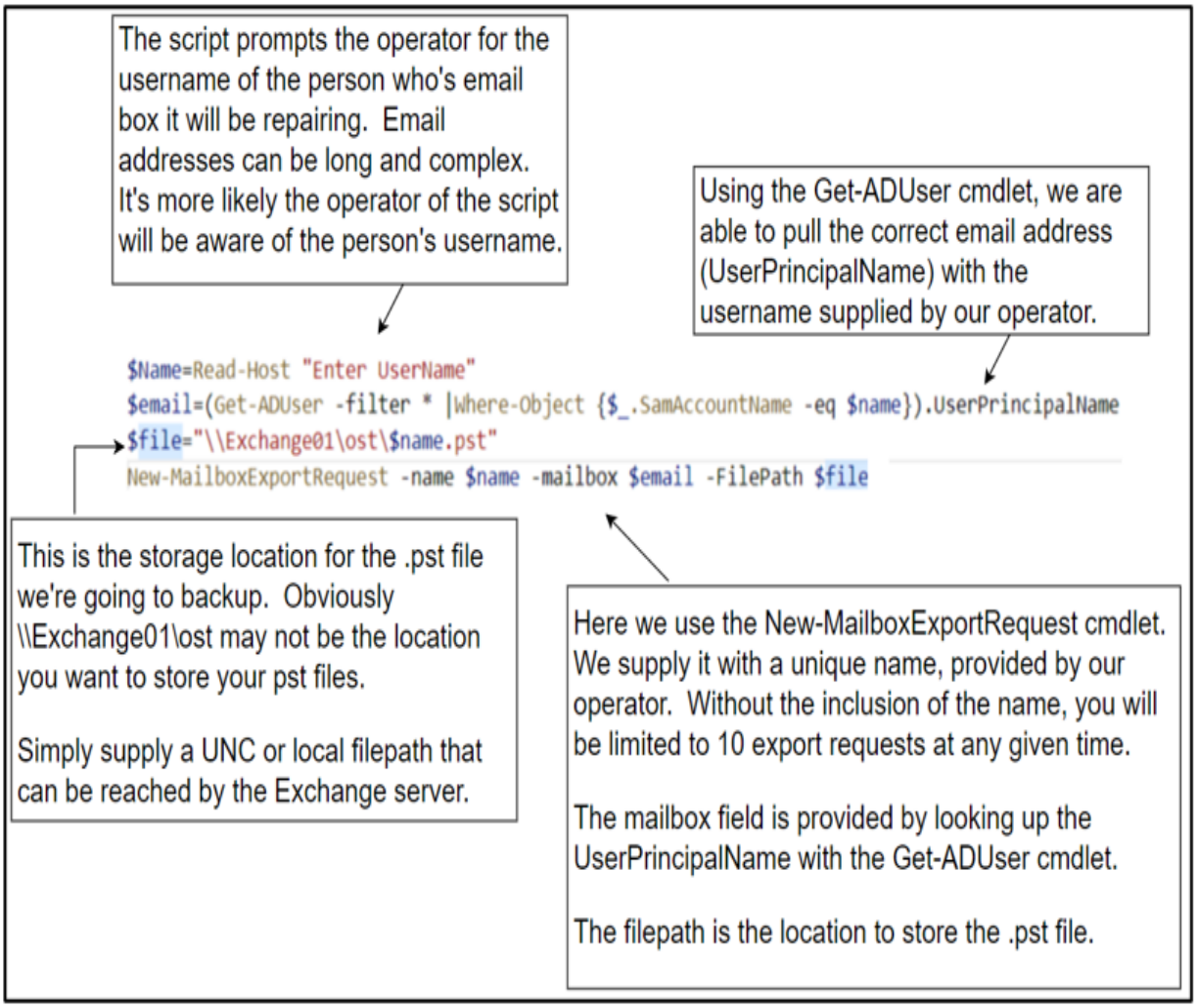
- New-MailboxExportRequest
- Get-MailboxExportRequest
- Disable-Mailbox
- Enable-Mailbox
- New-MailboxImportRequest
- Get-MailboxImportRequest

We will touch on how we can use these scripts to perform our mailbox repair. But our use case will only scratch the surface of what the Exchange modules are capable of. Let's see how we can utilize these specific Exchange cmdlets to repair a user mailbox with a single script. Then, if you're interested in learning more you can check out the further details of the Exchange modules at https://docs.microsoft.com/en-us/powershell/module/exchange.

7.4.1 New-MailboxExportRequest

New-MailboxExportRequest uses the native PowerShell backend to create a .pst copy of a user's mailbox. This is simply a way to back up an existing user's mailbox/calendars/contacts and folders (Figure 7.15).

Figure 7.15 New-MailboxExportRequest example.



7.4.2 Get-MailboxExportRequest

Now that we have sent the Exchange Server the request to back up a user's mailbox we can check the status of the request by utilizing the Get-MailBoxExportRequest. This will provide you details on where in the process the ExportRequest is. (It takes about 5 minutes for the average mailbox to complete this step. But extremely large mailboxes could take much longer).

There are a number of status that can be returned by this cmdlet. The typical statuses you might expect to see include:

- Queued

- InProgress
- Completed
- Failed

There are additional statuses that can be returned each of these can be found in the cmdlet's help file or online help sections. Since this is not a book that intends to deep dive into Exchange specific troubleshooting the four listed above are the statuses we will be focusing on.

The two statuses that we're interested in for our script are the "Completed" and "Failed". We have our do-until loop in our function to check for these two key statuses. If an Export is Queued we will simply wait, which our do-until loop does quite well. If it's InProgress, once again we are good to wait.

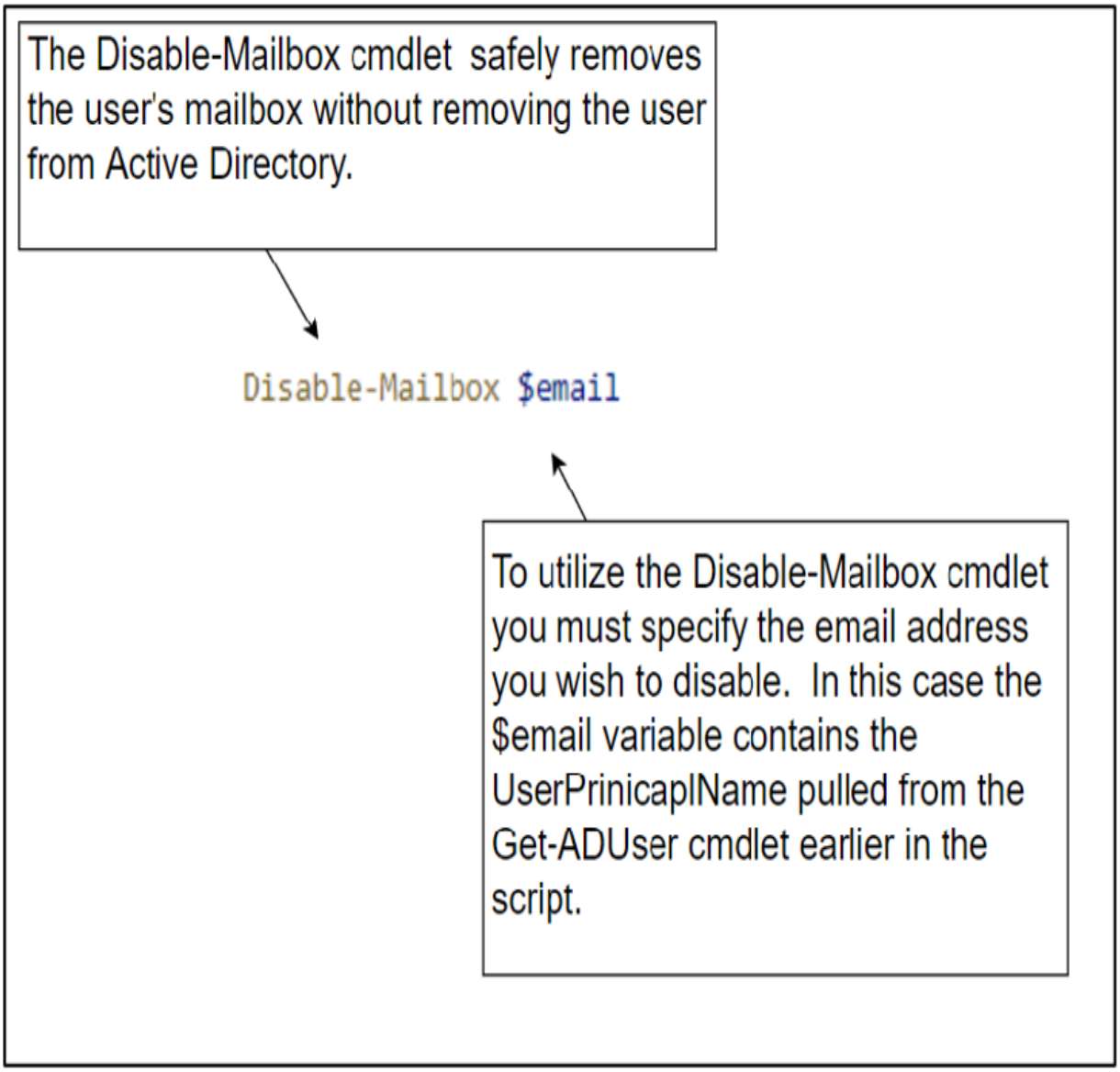
We only want to proceed if the status is "Complete"; however, there is always a chance something goes wrong. Because our script would automatically disable a user's mailbox we want to be very careful we don't complete this step without having the old .pst backed up.

Thus, we will also terminate our loop if it returns a "Failed" status. Our script is looking for this value and will write an error message to our user and end the script before any damage is done.

7.4.3 Disable-Mailbox

The Disable-Mailbox (Figure 7.16) cmdlet disables a current user's mailbox. The user account remains but the mailbox is no longer associated with the user's Active Directory attributes. The mailbox does still exist and can be re-connected by utilizing a ***Connect-Mailbox*** cmdlet.

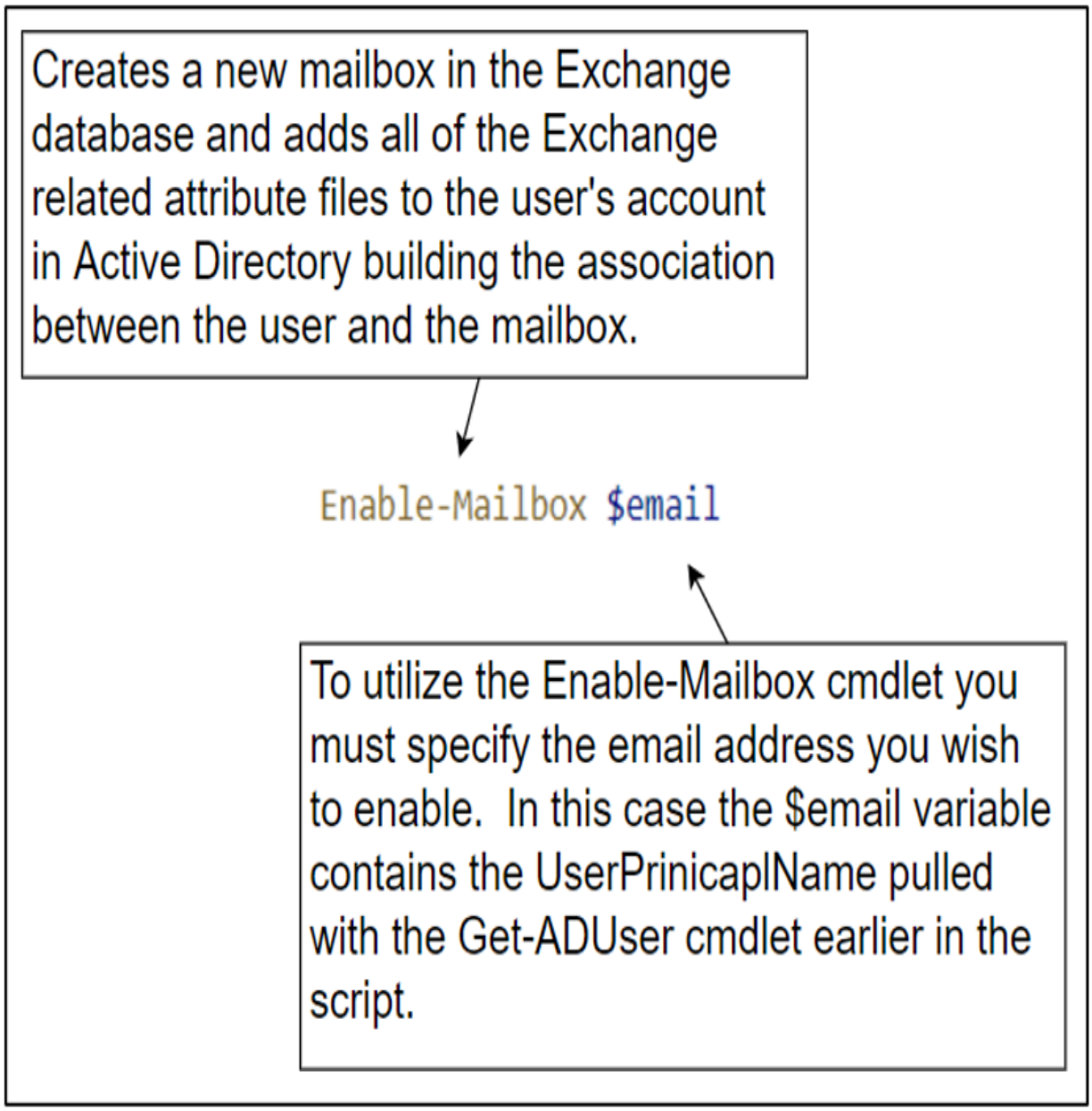
Figure 7.16 Disable Mailbox cmdlet



7.4.4 Enable-Mailbox

The Enable-Mailbox cmdlet enables an Exchange mailbox for an existing user without an associated email box (Figure 7.17). The cmdlet adds all of the Active Directory Exchange related attributes to the Active Directory account, but a new mailbox is created in the Exchange Database (thus the reason it often fixes broken email boxes associated with Exchange Database issues).

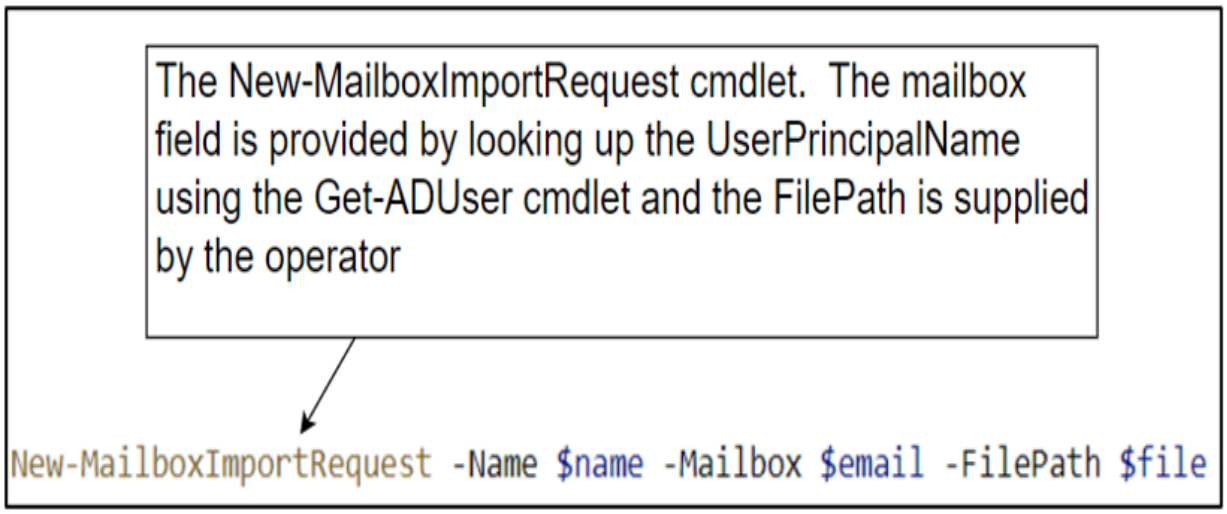
Figure 7.17 Enable-Mailbox cmdlet



7.4.5 New-MailboxImportRequest

The New-MailboxImportRequest command starts the process of importing a .pst file into an existing mailbox (Figure 7.18). With the old mailbox exported to a .pst file before disabling and enabling a new mailbox we can now copy over the contents of the .pst file bringing over all of the old mailbox's email, folders, calendar and contacts seamlessly into our new mailbox.

Figure 7.18 New-MailboxImportRequest cmdlet



7.4.6 Get-MailboxImportRequest

Much like when we used the MailboxExportRequest we will want a way to monitor the status of the import request. In general, a failure at this step is less of an issue than a failure on the automated backup process. Since the .pst file has already been backed up, if for some reason, the automated import fails, we can import manually.

The Get-MailboxImportRequest allows us to monitor the status of the ImportRequest. As before, there are many different statuses that are capable of being returned. But in general, we only are concerned with the same four as before.

- Queued
- InProgress
- Completed
- Failed.

If we get a failed status returned our code is set to inform us of the failure and tell us the location of the .pst created so the System Admin can then re-import the .pst file manually to the new mailbox.

7.5 Putting it all together

Here's how all of the tools work together to enable us to automatically repair a user's Exchange Database corruption with a script.

7.5.1 Create a Function

We create a function called "Get-Status" (Figure 7.19). This function accepts input for $user and $type.

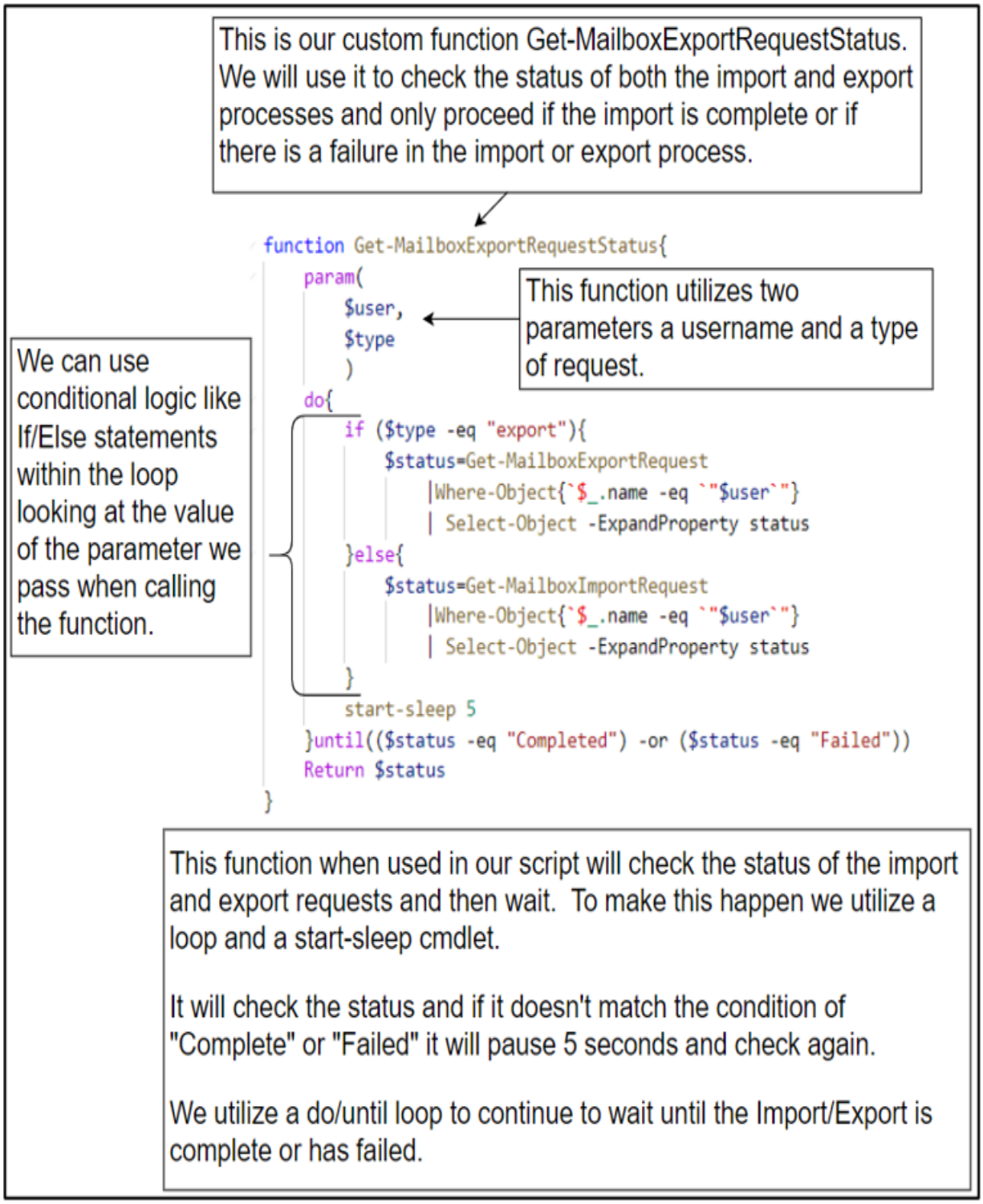
Depending upon the $type supplied it will run either a Get-MailboxImportRequest or a Get-MailboxExportRequest to automatically check the status of our import/export attempts.

It utilizes a new loop function called a **do-until** loop. The loop checks for the status and then waits until the status is either "Completed" or "Failed". The function then returns the "Completed" or "Failed" result once the backup/import is complete.

Utilizing a function saves us from having to type the code twice (once for backup and once for import). It also makes the script easier to update because if we later make a change to this; we only have to change it once to affect any number of times we later utilize the function.

Because PowerShell is an interpreted language, our function needs to be defined before we try to use it. Thus, it is good practice to write all of your functions at the beginning of your script regardless of when you will later use them in your actual script.

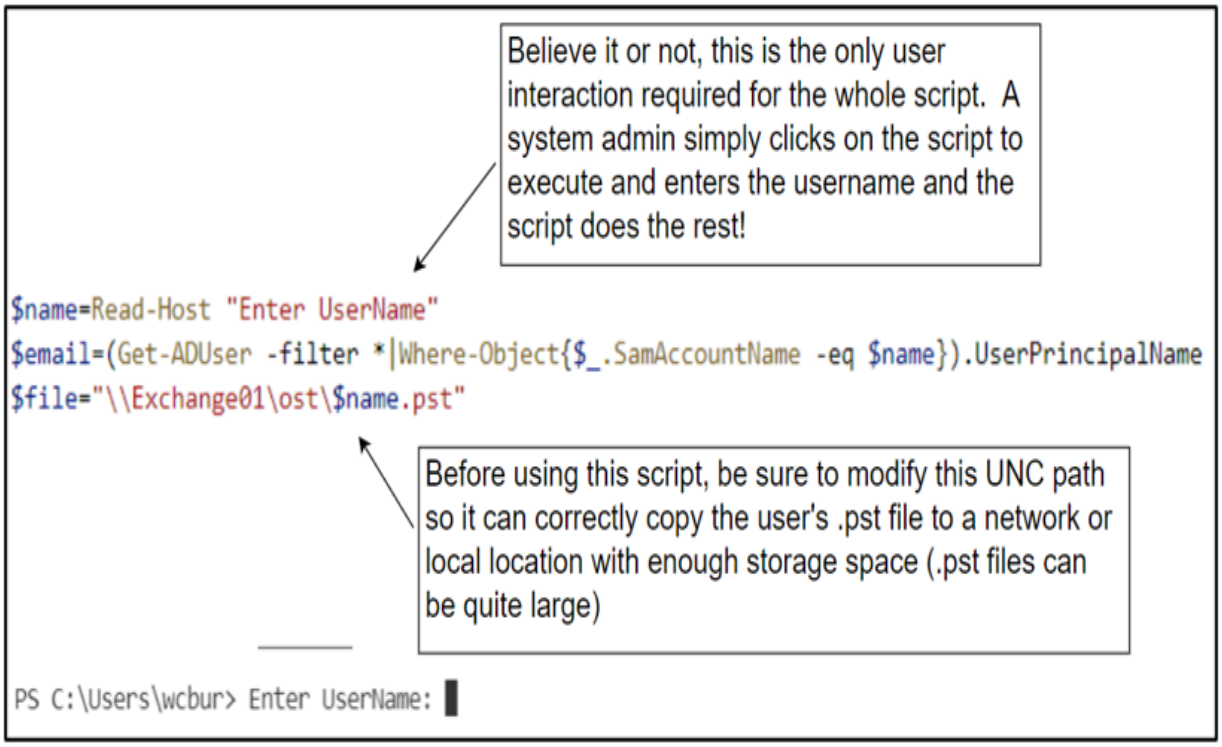
Figure 7.19 Tiny PowerShell code get-status function.



7.5.2 Get User Information

Utilizing a ***Read-Host*** cmdlet we prompt the operator of the script to supply us with the login name of the user we will be repairing. Through the use of the ***Get-ADUser*** cmdlet we are then able to make sure we are repairing the correct email address (Figure 7.20). Email addresses have a tendency to be long and users with similar names may have very similar email addresses. The inclusion of the username helps assure the operator that we are repairing the correct mailbox.

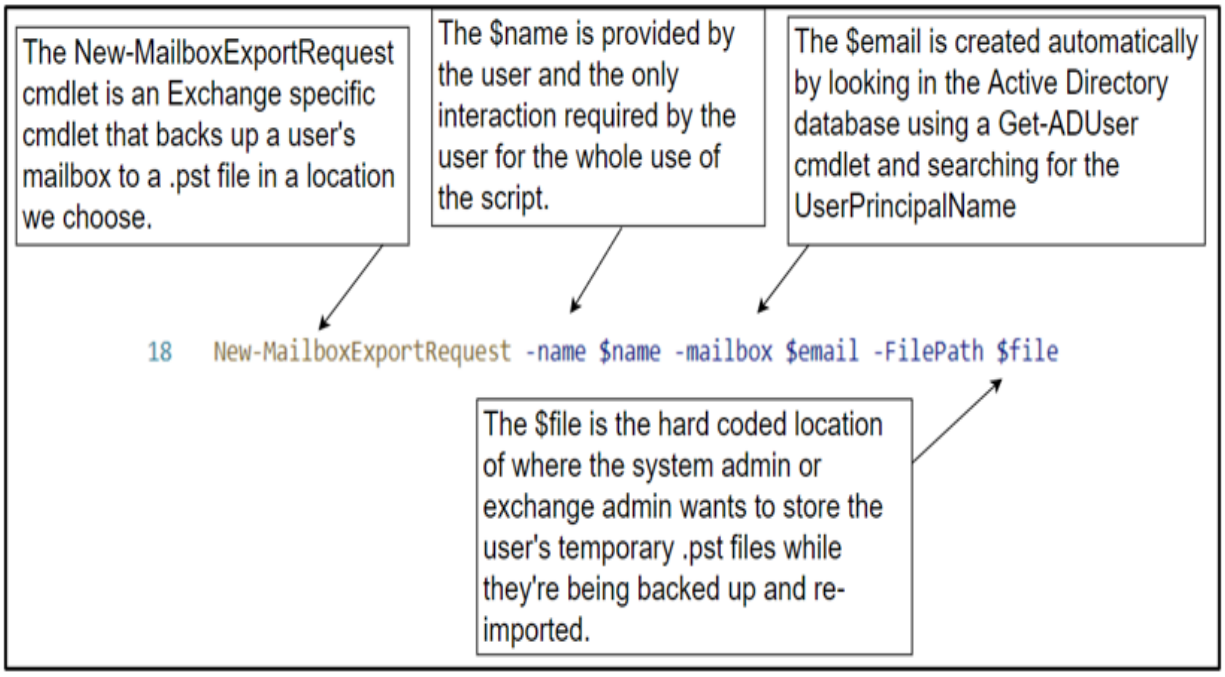
Figure 7.20 Get user information from Tiny PowerShell project code



7.5.3 Backup User's Mailbox

Using Exchange specific cmdlets we are able to issue a ***New-MailboxExportRequest*** to the Exchange server for the user provided (Figure 7.21). This creates a backup .pst file saving the user's inbox/folders, calendars, and contacts into a .pst folder.

Figure 7.21 New-MailboxExportRequest from Tiny PowerShell project code



7.5.4 Wait for Backup to Complete

With the Function we created we are now able to have PowerShell watch the status of the Export Request with the use of the ***Get-MailboxExportRequest*** (Figure 7.22)and waiting for the status to read either completed or failed. PowerShell will re-check this status automatically every 5 seconds without further operator interaction. As soon as one of those status are returned the script continues automatically (Figures 7.23, 7.24, and 7.25).

Figure 7.22 Get-MailboxExportRequest process detailed (part 1)

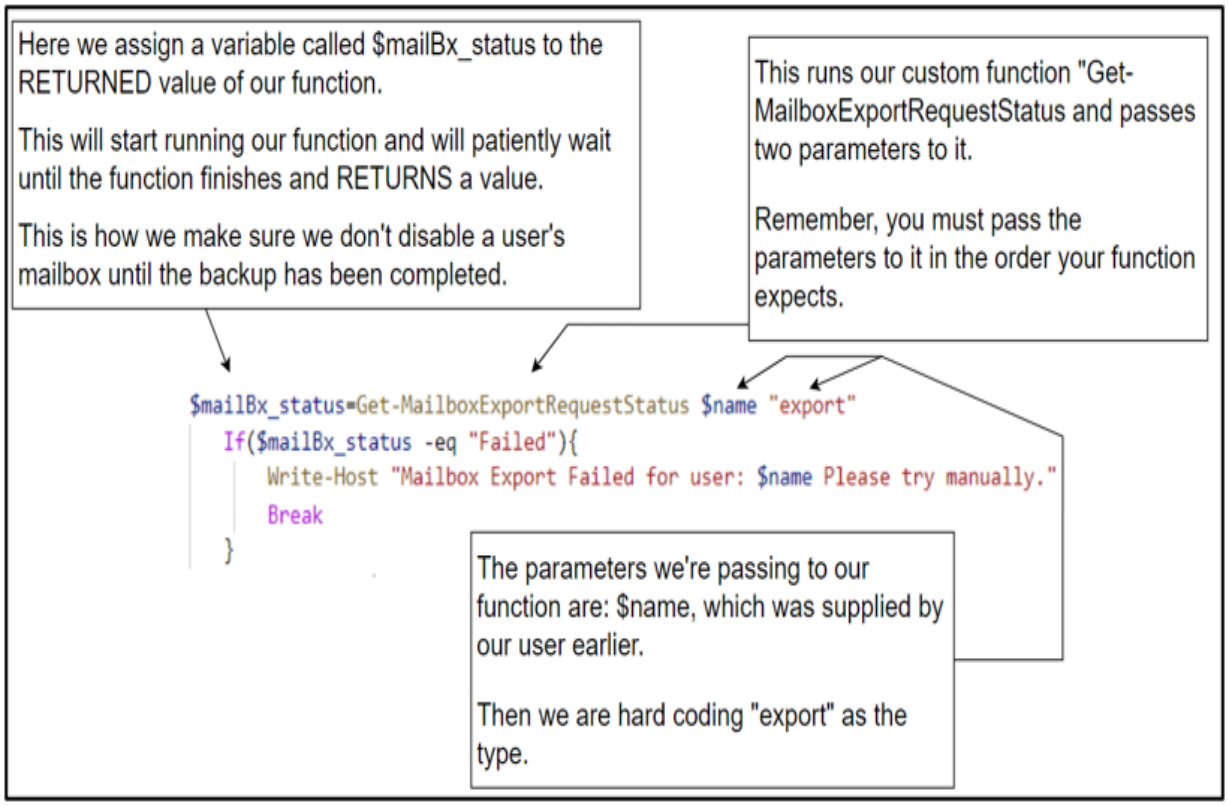


Figure 7.23 Get-MailboxExportRequest process detailed (part 2)

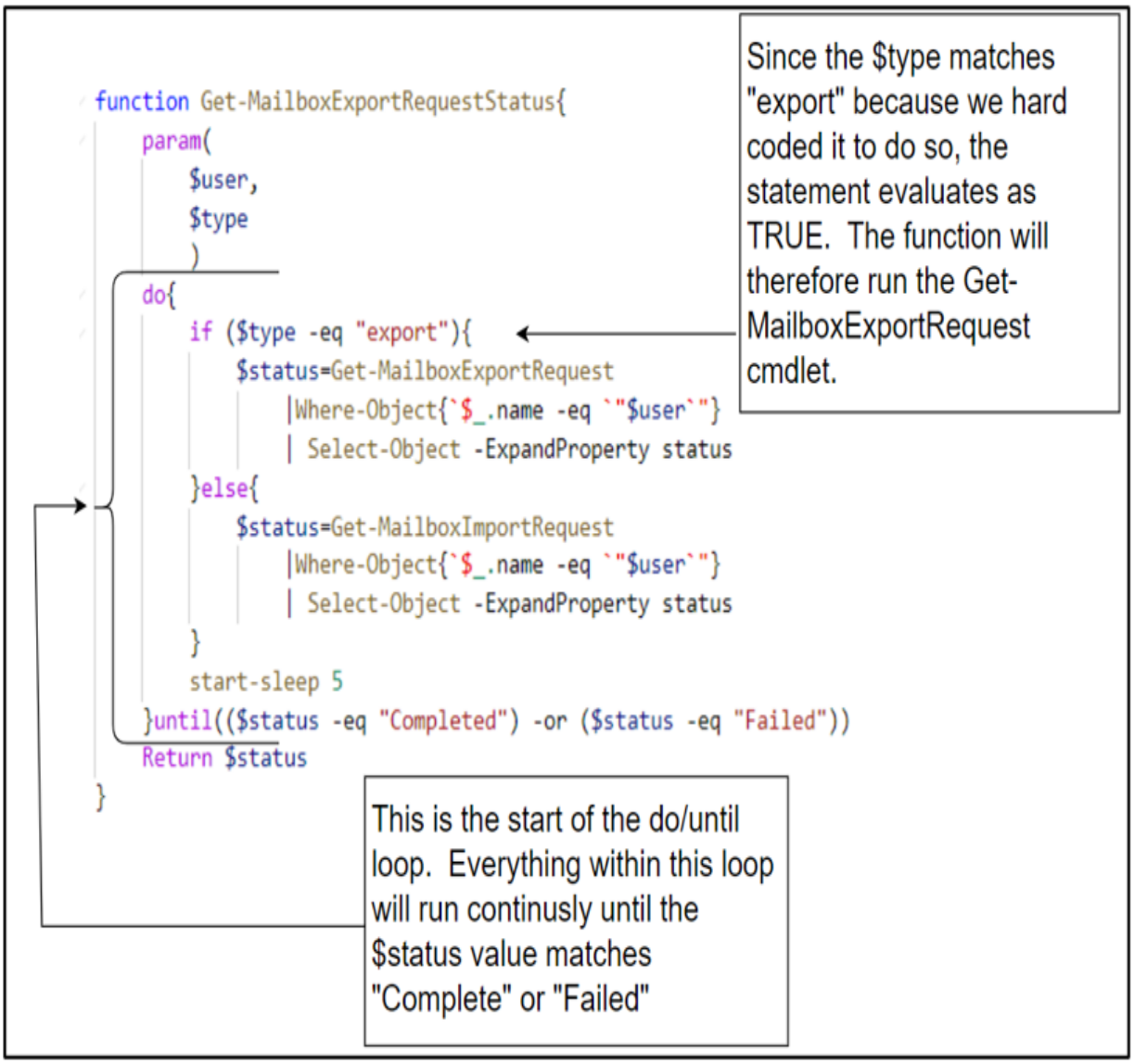


Figure 7.24 Get-MailboxExportRequest process detailed (part 3)

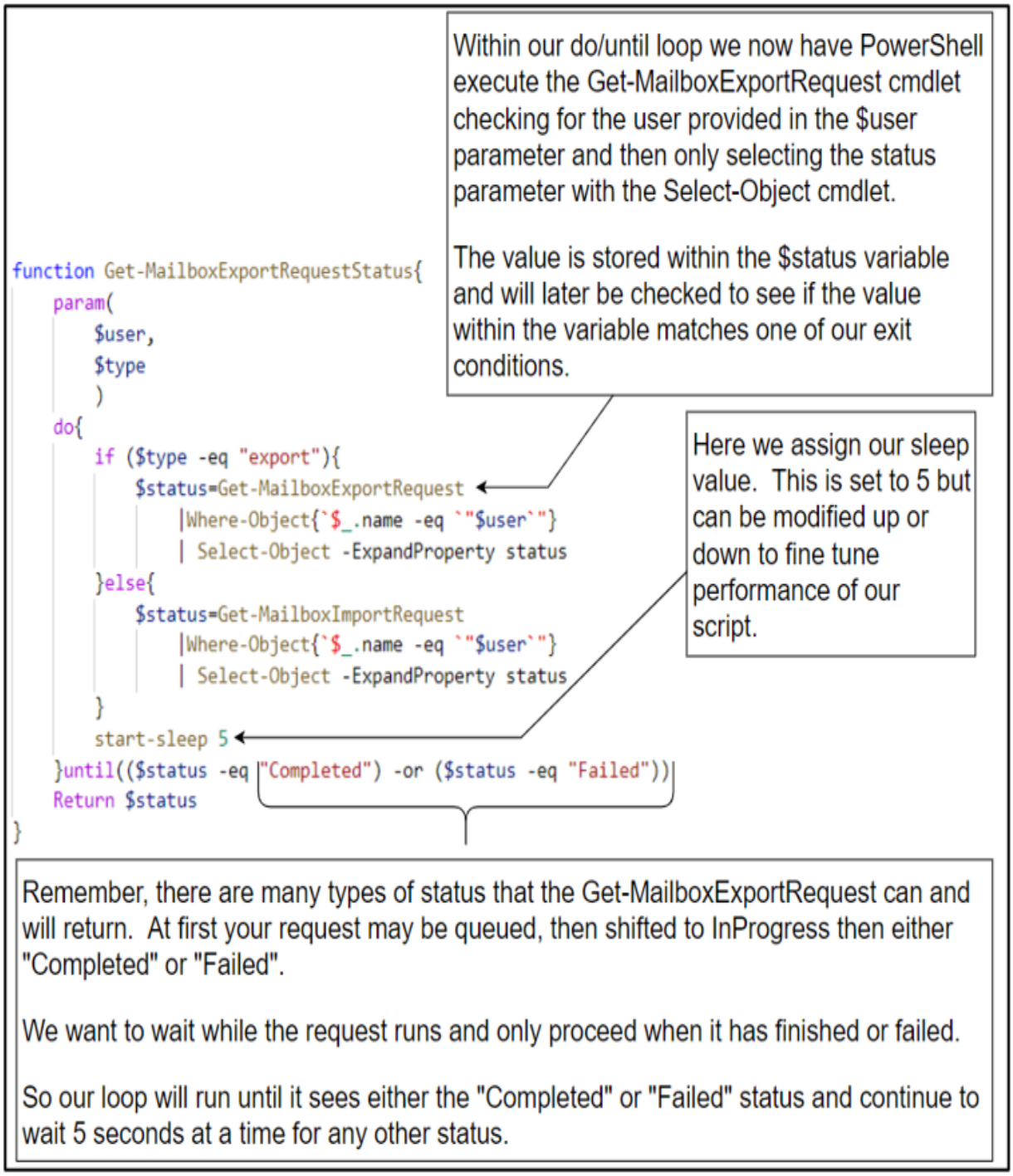
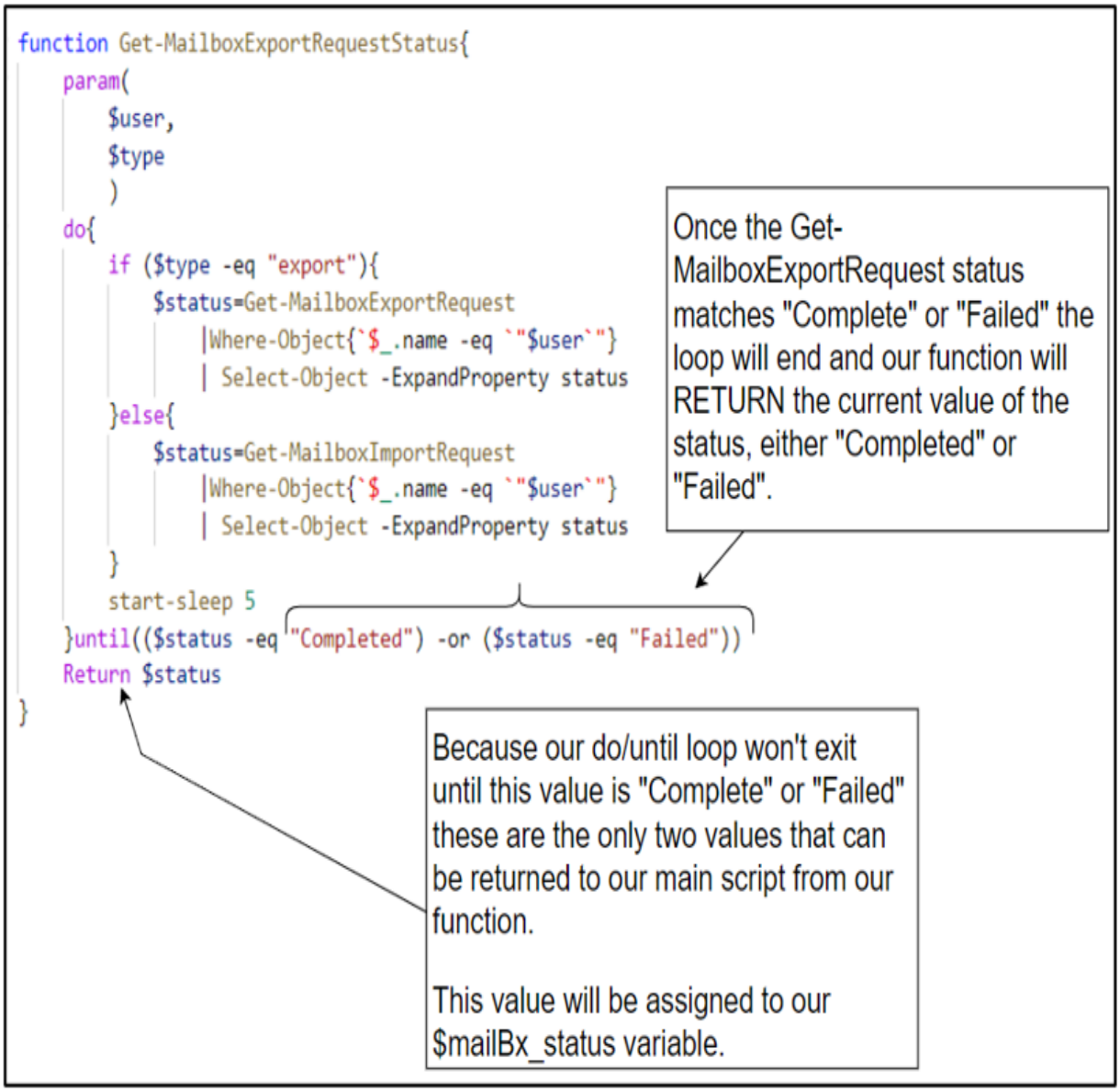
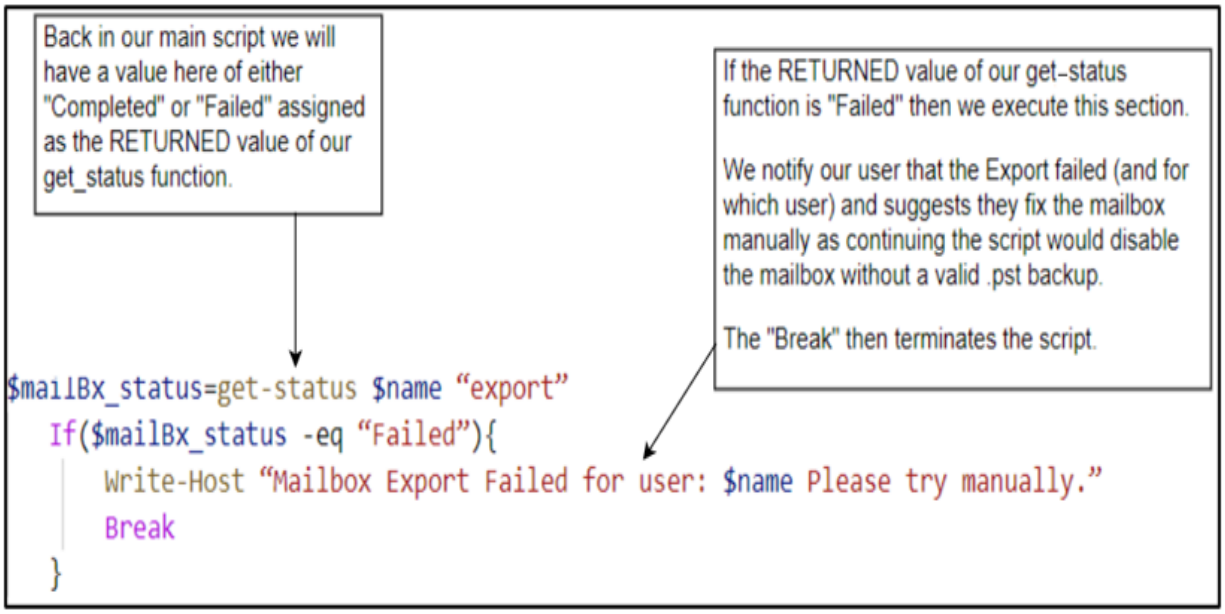


Figure 7.25 Get-MailboxExportRequest process detailed (part 4)



If the **Get-MailboxExportRequest** returns a failed state the script detects this and displays a message to the operator that it was unable to back up the user's mailbox before any data is lost (Figure 7.26).

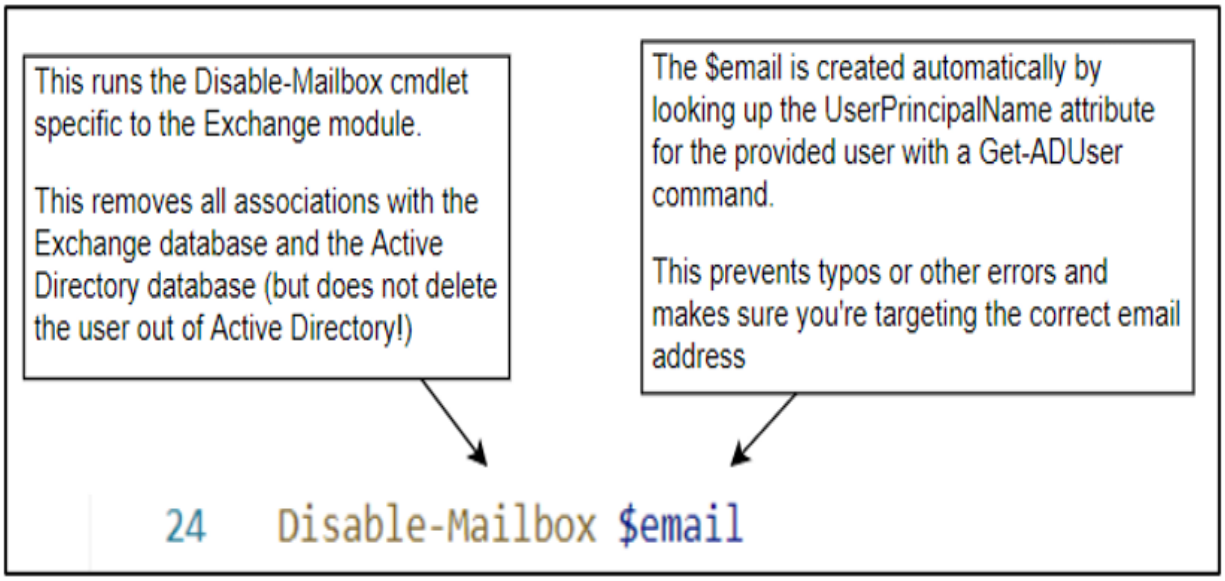
Figure 7.26 Get-MailboxExportRequest process detailed (part 5)



7.5.5 Disable Bad Email Box

With the old mailbox safely backed up, the script then disables the bad mailbox utilizing the ***Disable-Mailbox*** cmdlet (Figure 7.27). This does not affect the user's account but removes any association between the account and the mailbox. The mailbox becomes disconnected, and all Exchange related attributes are removed from the user's account in Active Directory.

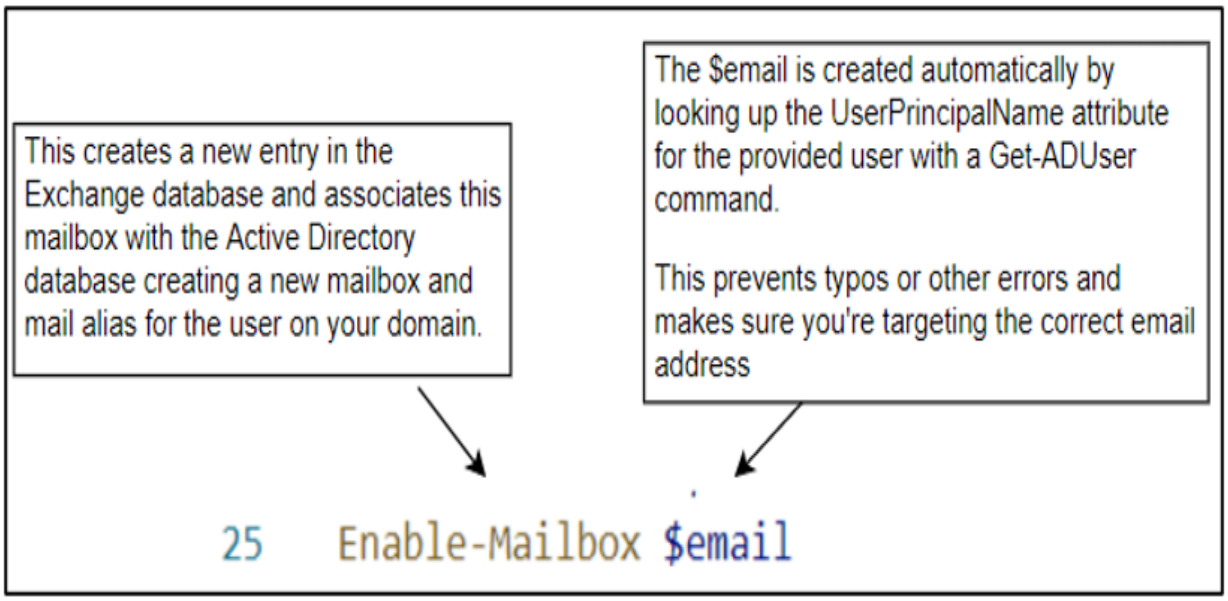
Figure 7.27 Disable Mailbox from Tiny PowerShell project code



7.5.6 Enable New Mailbox

Then a new mailbox is created in the Exchange Database for the existing user (Figure 7.28). The new mailbox is then associated with the user's Active Directory account and all Exchange Attributes are automatically set for this new mailbox.

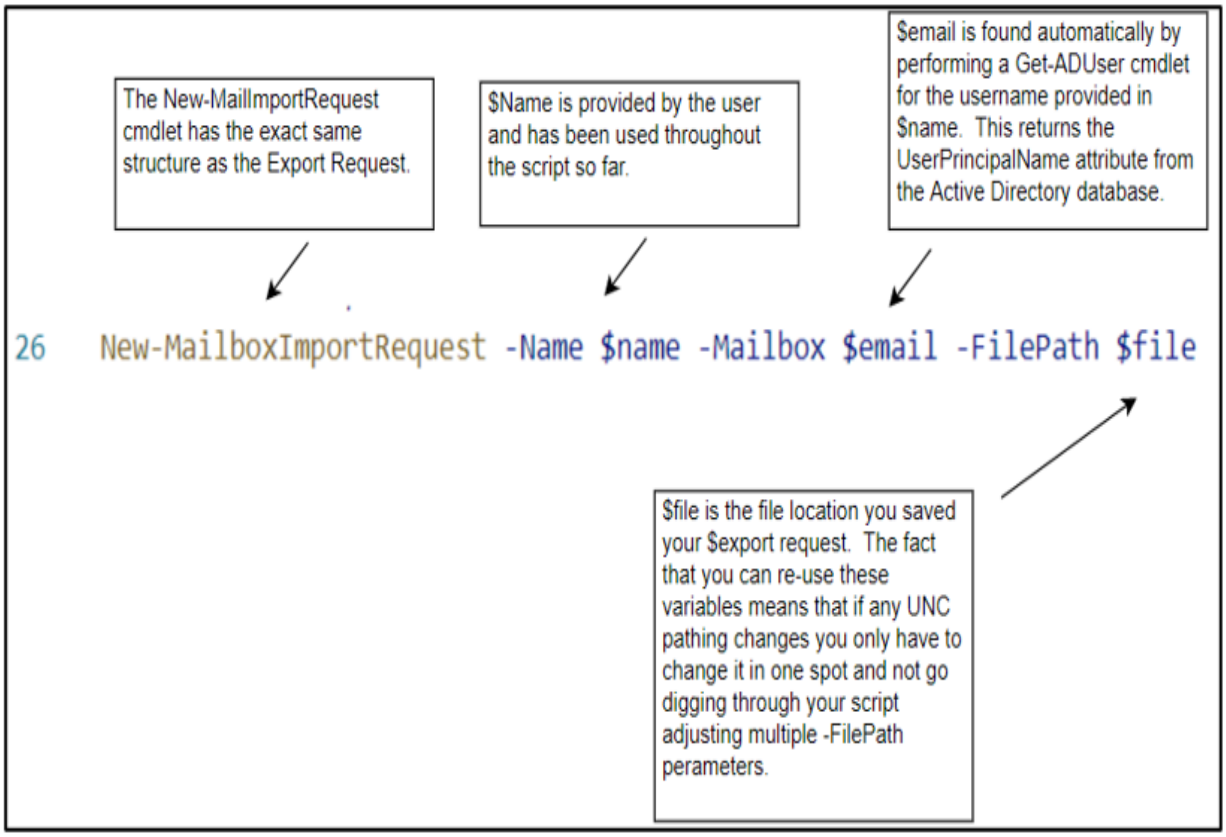
Figure 7.28 Enable Mailbox from Tiny PowerShell project code



7.5.7 Import Backup from Old Mailbox

The .pst file from the old mailbox is then imported into the new mailbox utilizing the ***New-MailboxImportRequest*** cmdlet (Figure 7.29). This ensures that the new mailbox now includes all of the old mailbox's email, folders, calendars and contacts for a seamless transition.

Figure 7.29 New-MailboxImportRequest from Tiny PowerShell project code



7.5.8 Wait for Import to Complete

Once more utilizing the function we created we are able to monitor the status of the mailbox import request using the ***Get-MailboxImportRequest***. The function once again checks every five seconds for the status to change to "Completed" or "Failed" (Figures 7.30, 7.31, and 7.32).

Figure 7.30 Get-MailboxImportRequest process detailed (part 1)

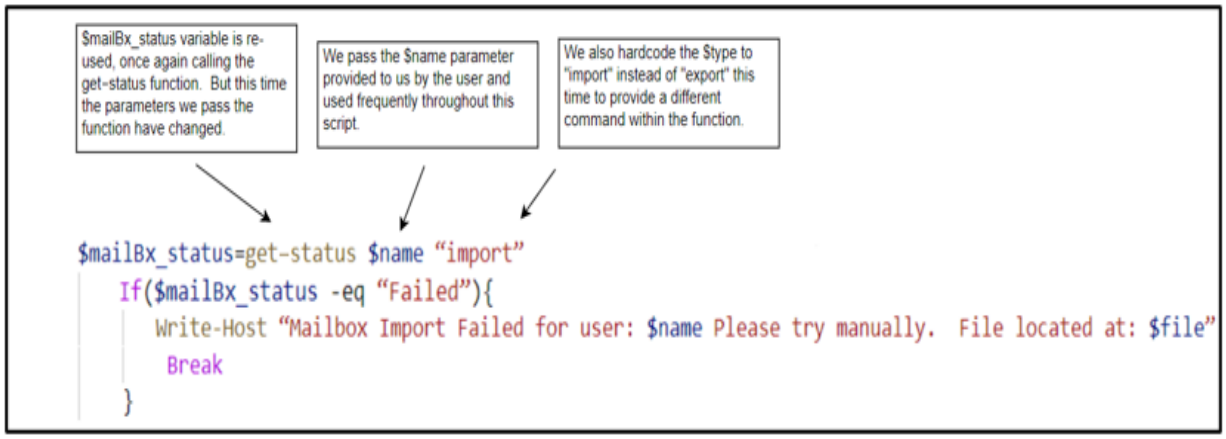


Figure 7.31 Get-MailboxImportRequest process detailed (part 2)

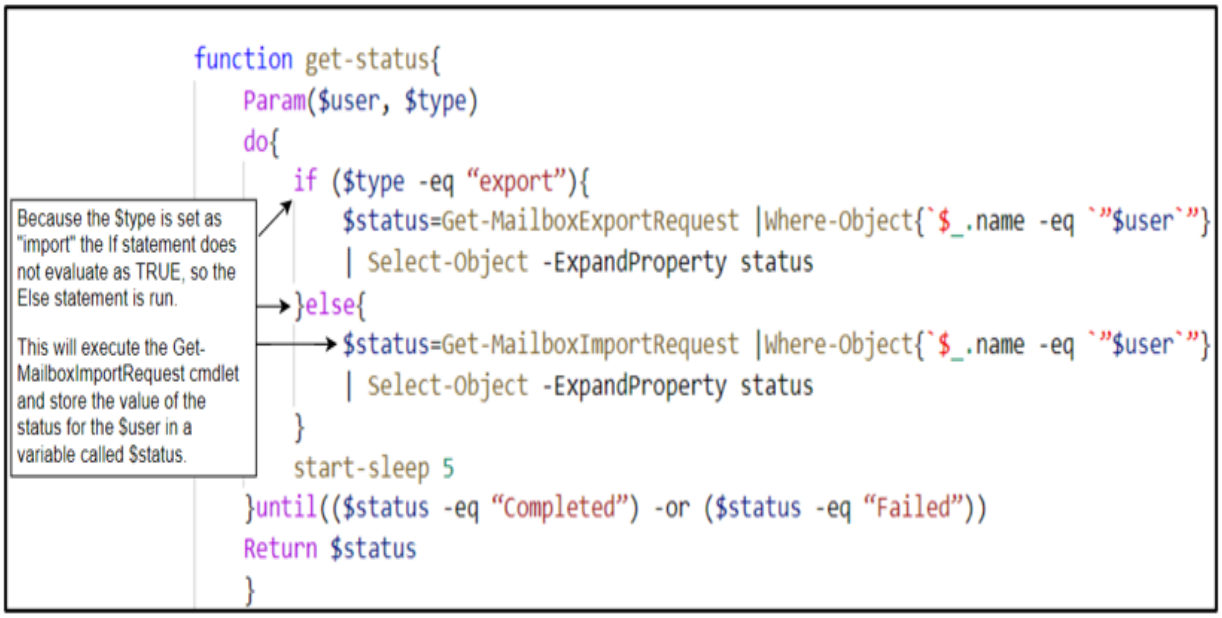
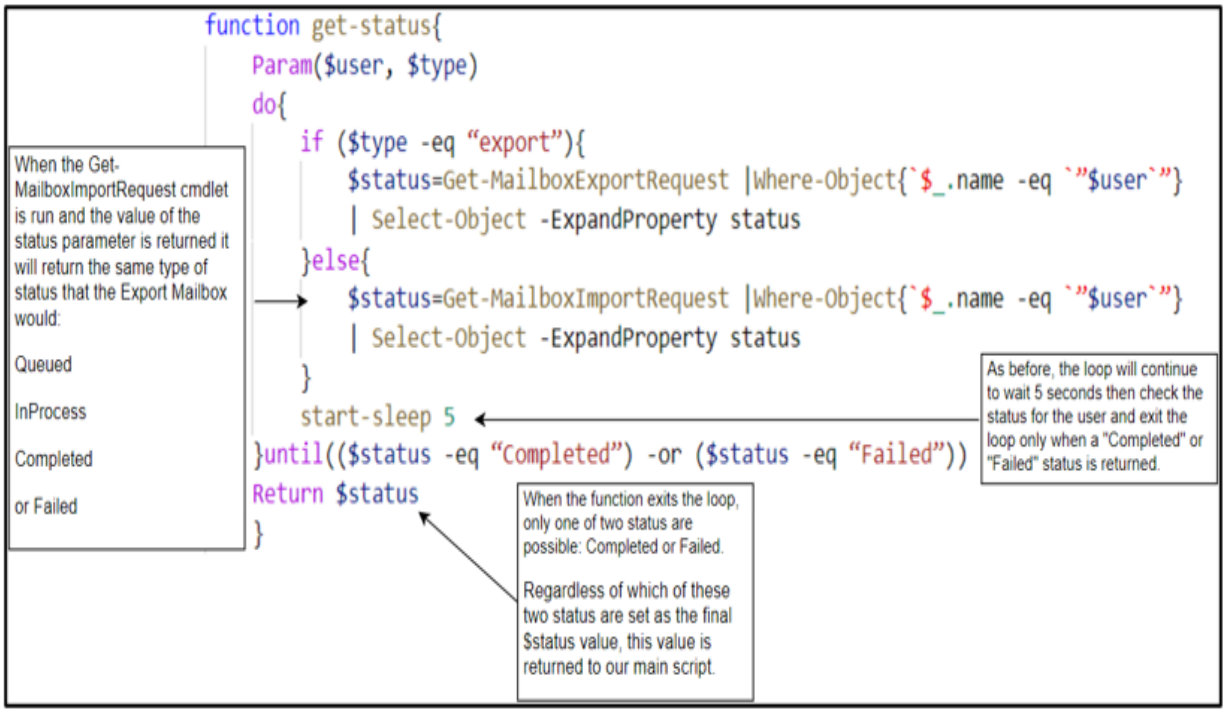
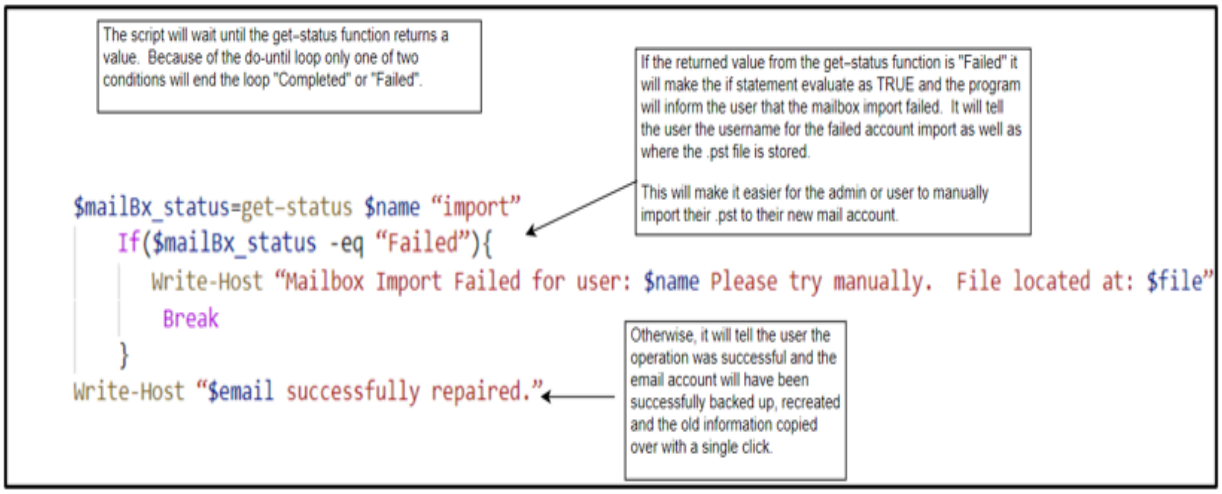


Figure 7.32 Get-MailboxImportRequest process detailed (part 3)



If a "Failed" message is returned, the script informs the operator that the import was unsuccessful and provides the operator the path to the .pst that was created earlier to make a manual attempt at a .pst file import simpler (Figure 7.33).

Figure 7.33 Get-MailboxImportRequest process detailed (part 4)



7.6 Summary

- Functions are a list of PowerShell statements that are assigned a name. These lines of code can then be called in your Main script once or multiple times.
- Functions can save a lot of repetitive code in your script by just writing it once and calling it each time you want to perform those statements.
- Functions can be simple or very dynamic; capable of loops and conditional logic that help you branch your code to make a single function perform a variety of actions.
- There are many types of loops used in PowerShell.
- While loops all do the same thing, running PowerShell statements repeatedly on one or more targets, different types of loops can be run to determine targets and exit conditions.
- A new type of loop is known as the do-until loop, where the statements executed on targets until a specified exit condition is achieved through the evaluation of conditional logic.
- Conditional logic does not need to be a singular statement but can include multiple evaluations with the use of the -or within the condition.
- There are times when constructing complex strings you need specific key characters to be ignored. You can do this with an escape character. PowerShell uses the backtick (`) as an escape character.
- When you load the Exchange Module in your PowerShell session you can access to many Exchange related cmdlets. We used the New-ExportMailboxRequest, New-ImportMailboxRequest, Get-MailboxImportRequest, Get-MailboxExportRequest, Disable-Mailbox, and Enable-Mailbox cmdlets to backup a user's mailbox information, repair and then import the old data back.
- The New-ExportMailboxRequest and New-ImportMailboxRequest cmdlets export a user's mailbox information into a .pst file and then import this .pst file to a new mailbox, respectively.
- The Get-MailboxImportRequest and Get-MailboxExportRequest cmdlets check the current status of mailbox import request and export request, respectively. They return one of many statuses including, but not limited to, Queued, InProgress, Completed, or Failed.
- The Disable-Mailbox cmdlet removes the association with the Exchange mailbox and database from the user's Active Directory user account and database.

- The Enable-Mailbox cmdlet creates and enables a mailbox in Exchange and makes all of the associations in the Active Directory user account and database for use on the Domain.

8 Unified user creation

This chapter covers:

- Using conditional logic to call a function.
- Using Exchange cmdlets on any PC in your domain utilizing PSSessions.
- Creating unique usernames to prevent duplicates
- Unifying several scripts to create a user, email and default groups with a few questions.
- Testing and optimizing your searches.

Up until now we have dealt with scripts that do very specific things. They might help us create an email address, create a user, or transfer group memberships. Each of these scripts are useful in their own right, but we're interested in automation. Why not create one script that does all of these things?

As we have seen, each of these scripts are a time savings in themselves. It can take several minutes to create an Active Directory user by hand. It can take dozens of clicks to duplicate group membership from one user to another. Each of these scripts deserves a place in our ecosystem of scripts. But one of the core concepts of automation is doing more with less. If we could integrate some of the things we've done and weave them into a coherent script that did all of those things; that would save us even more! Here we can create a unified user creation script.

This script will:

- Check to see if we have established a connection to the Exchange services on our domain. If not, it creates one, we can import Exchange cmdlets to any PC on the domain utilizing the PSSession concepts.
- With the use of a mandatory parameter we prompt the user for First name, Last name and a user SamAccountName to copy groups from.

- Creates a username that is no more than 8 characters long. Most domains should have usernames limited to 8 characters or less. This is especially important in domains that have both Windows and Unix and it traditionally and historically best practice.
- Checks to see if the username is already in use. Many names are common. It is not at all uncommon to have several people with the same last name working in even moderately sized enterprises. Utilizing a first initial and last name combination for a username may return duplicates. James Smith and Jason Smith for example would both be jsmith. This script has a way to detect that a username is already taken and increment the new one by numbers. So, if James Smith already existed Jason Smith would be created as jsmith1.
- Creates the user in active directory, sending the information to a log file.
- Prompts the user for a username to copy groups from. With Role Based Access Controls (RBAC) and the concepts of Least Privilege, it's generally best practice to make sure that only users who need access to a resource on your domain have access. This can efficiently be done by utilizing Active Directory Groups. But it can be frustrating for new employees and their management to have to discover all of these groups by trial and error. By using template accounts, however, you can set up all of the basic groups needed by a role and grant these groups to all new users needing that role. In this way, the new person in HR, for example, has all of the basic HR required groups as soon as the account is created, making them more efficient from day one.
- The script then uses the Exchange specific cmdlets to create an email address and link the Exchange database to the Active Directory database automatically.
- User creation, Group Membership and Email creation are all logged into individualized log files; should any errors occur exactly what happened will be stored there.

8.1 Project Code

This is the first of our scripts too large to be in a single screenshot! The code is below, and we'll be exploring this code in detail through the rest of this chapter.

```
param (    #A
    [Parameter(Mandatory=$true)]    #B
    $FirstName,    #C
    [Parameter(Mandatory=$true)]
    $LastName,
    [Parameter(Mandatory=$true)]
    $CopyFromUserSamAccountName
)    #D
function Connect-Exchange {
    param (
        $ExchangeServer,
        $UserCredential
    )
    $Session = New-PSSession -ConfigurationName
Microsoft.Exchange `    #E
        -ConnectionUri http://$ExchangeServer/PowerShell/ `
        -Authentication Kerberos -Credential $UserCredential
    Import-PSSession $Session -DisableNameChecking    #F
    Return $Session
}
$ExchangeServer="exchange01.ForTheITPro.com"
if (-Not(Get-PSSession|Where-Object {$.ConfigurationName -eq
#G
"Microsoft.Exchange"})){
    Write-Output "Exchange Module not loaded."
    Write-Output "Connecting Exchange…"
    $UserCredential = Get-Credential    #H
    $Session=Connect-Exchange $ExchangeServer $UserCredential
}
$LastLength=$LastName.Length
if ($LastLength -ge 8){
    $LastName=$LastName.Substring(0,7)
}
$UserName=$FirstName.Substring(0,1).ToLower()+$LastName.ToLower(
)
$dup=Get-ADUser -filter {SamAccountName -eq $UserName}
if ($null -ne $dup){    #I
    Write-Output "Duplicate User."
    $i=1
    do{
        $TestName=$UserName+$i
        $dup=Get-ADUser -filter {SamAccountName -eq $TestName}
        $i++
    }until ($null -eq $dup)
$UserName=$TestName
    if ($UserName.Length -ge 8){
        $UserName=$UserName.Substring(0,8)
    }
```

```
}
$LogFile="C:\Logs\$UserName.log"
Write-Output "Creating user: $UserName"|Tee-Object $LogFile -
Append
New-ADUser -Name $UserName -GivenName $FirstName -Surname
$LastName `
    -DisplayName $UserName -SamAccountName $UserName -Enabled
$false
    |Tee-Object $LogFile -Append
Start-Sleep 5
$Groups=(Get-ADUser -Identity $CopyFromUserSamAccountName -
Properties MemberOf).MemberOf
$Groups|Add-ADGroupMember -Members $UserName
Write-Output "$UserName added to: $Groups"|Tee-Object $LogFile -
Append
(Get-ADUser -Identity $UserName).SamAccountName|Enable-Mailbox
Get-PSSession|Where-Object {$_.ConfigurationName -eq
"Microsoft.Exchange"}
    |Remove-PSSession $Session    #J
Write-Output "$(Get-Date) $UserName Account successfully
created."|Tee-Object $LogFile -Append
```

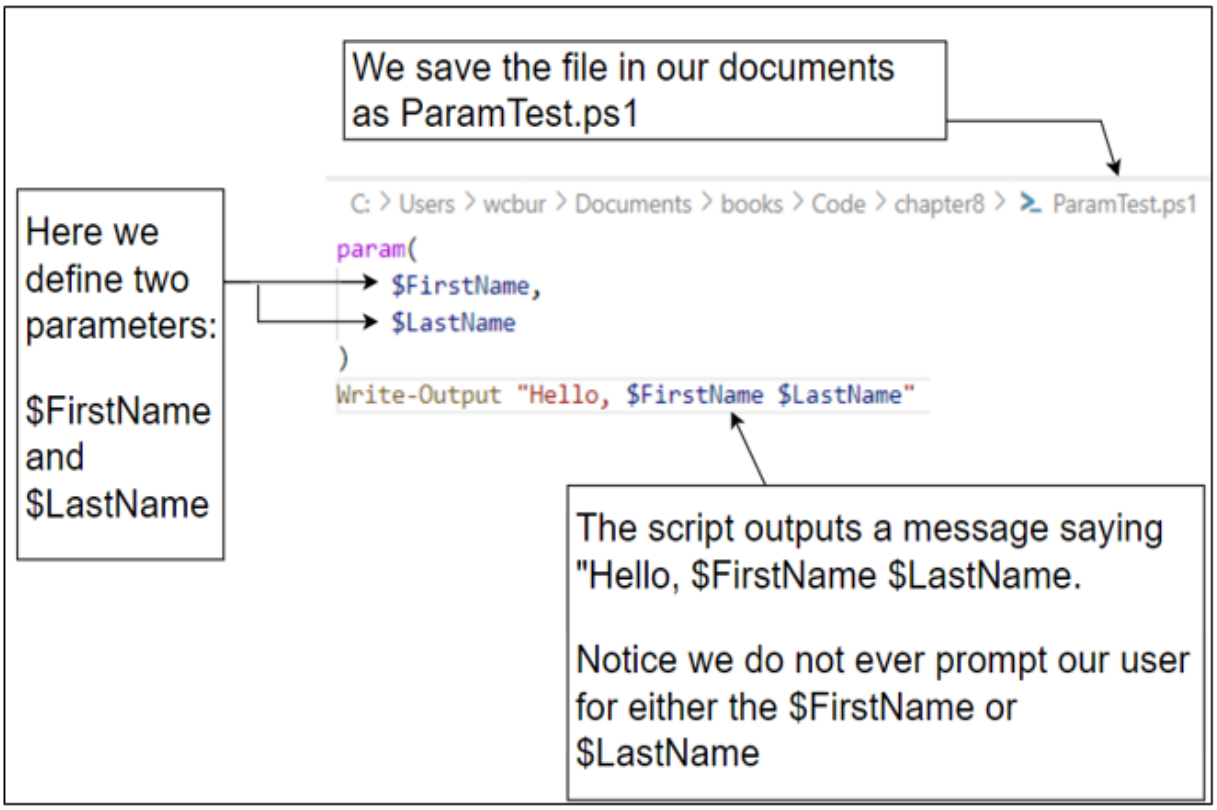
8.2 Script Parameters

While it is great we can prompt our user to supply us information using the Read-Host cmdlet, it still requires manual interaction. Often this is what we like, but wouldn't it be more efficient if we could simply supply all of those variables we prompt our user for, ahead of time?

This is exactly what happens when we parameterize our code. A Parameter, in programming terms, is a named variable we can pass into a script or function.

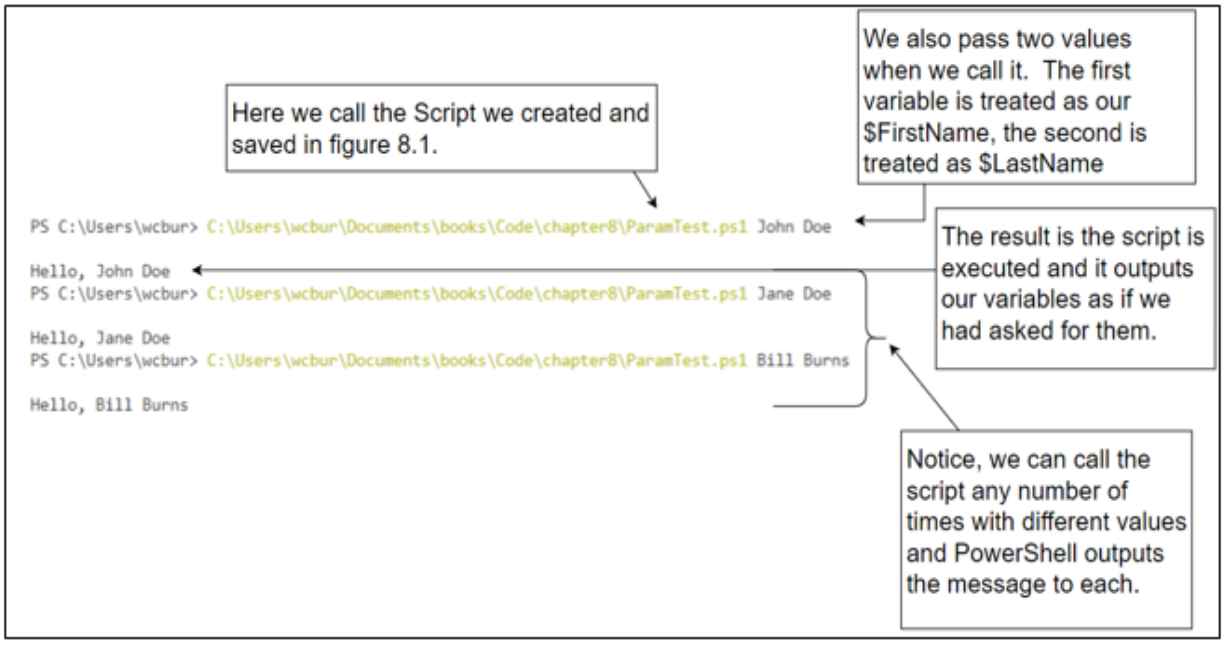
To define a parameter in a script we simply pass PowerShell the param keyword followed by one or more defined parameters contained within the parenthesis.

Figure 8.1 Parameters in a script.



When we later call the script, we pass it the parameters it is expecting, and PowerShell interprets the rest. When we call the ParamTest.ps1 script we just created we include a first and last name and PowerShell will construct our Output statement.

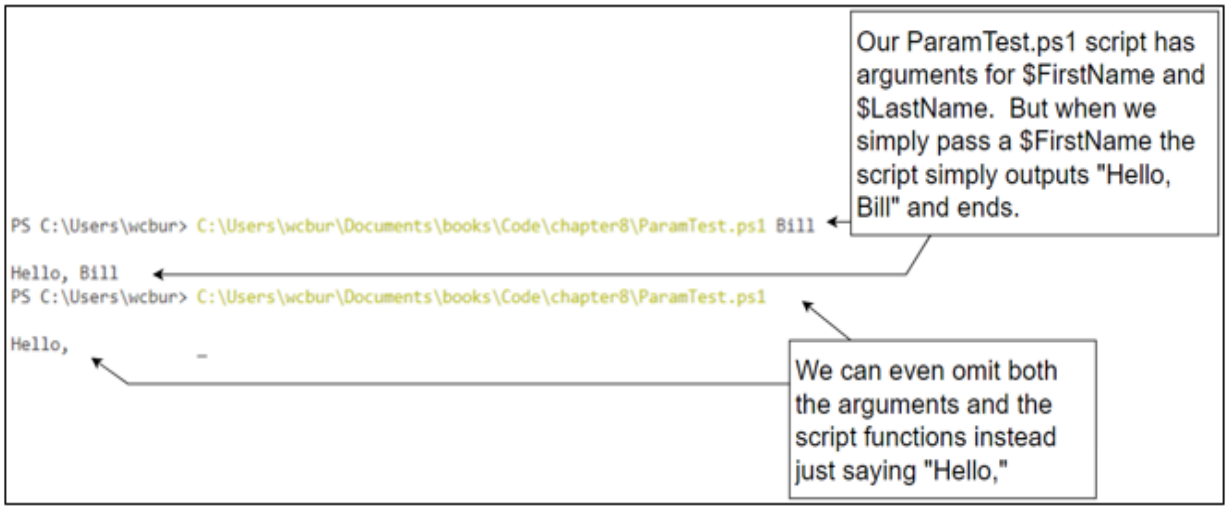
Figure 8.2 Calling a script with parameters.



8.2.1 Mandatory Parameters

There are times when we pass parameters into our functions or scripts that omitting them causes no problems or issues. When we look at the script we created in Figure 8.1 forgetting to pass a $LastName doesn't cause any real error. PowerShell simply has a NULL value for $LastName and passes the NULL value into the Output. We can even omit all of the variables and PowerShell passes both variables as NULL in the output.

Figure 8.3 Execution with a Non-Mandatory Parameter



If, however, like in our script the variable is required for the script to function we need a way to make sure that the variable is included when the function or script is called, or that we prompt our user for it before the script is executed.

This is known as a Mandatory Parameter. In our unified user creation script we create usernames, Active Directory accounts and email addresses based upon the first and last name supplied to us by our user. If we do not include a variable the script is expecting our script will fail to run, or perhaps worse run with unintended results. This can cause havoc in a production environment.

Fortunately, there are guards we can put on our scripts that will help us control for these unintended consequences. The first of these guards is the Mandatory attribute within the param keyword.

When we use the param keyword to set the Parameter attribute on our script or function we can set a Mandatory attribute to tell PowerShell if our expected parameter(s) are mandatory, or if, like in the script above, they can safely be ignored.

The syntax for this is:

```
[Parameter(Mandatory=$true)]
```

Followed by the coma separated name(s) of the parameters.

```
param (
    [Parameter(Mandatory=$true)]
    $FirstName,
    [Parameter(Mandatory=$true)]
    $LastName
)
```

Because of this granularity we can specify a number of parameters that are mandatory and leave other parameters as optional.

When a mandatory parameter is omitted, the script automatically prompts the user for the parameter. After it is entered the script runs as expected. This way, you can make all of your required inputs as Mandatory and even if the user omits them, or is unaware the parameter is required; PowerShell will prompt the user for the parameter. This works best when the parameters are well named and simple to understand. Having the user understand you're expecting a $FirstName is a lot more intuitive than $Parameter1.

Figure 8.4 Naming Mandatory Parameters.

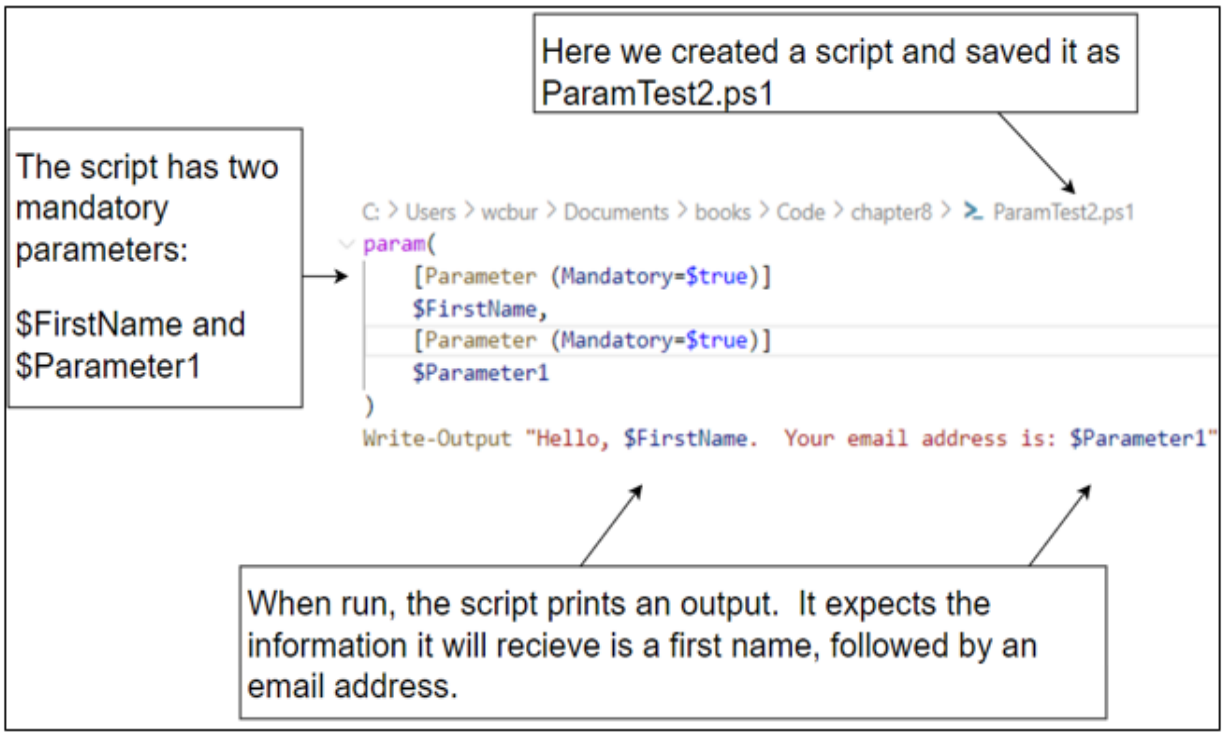
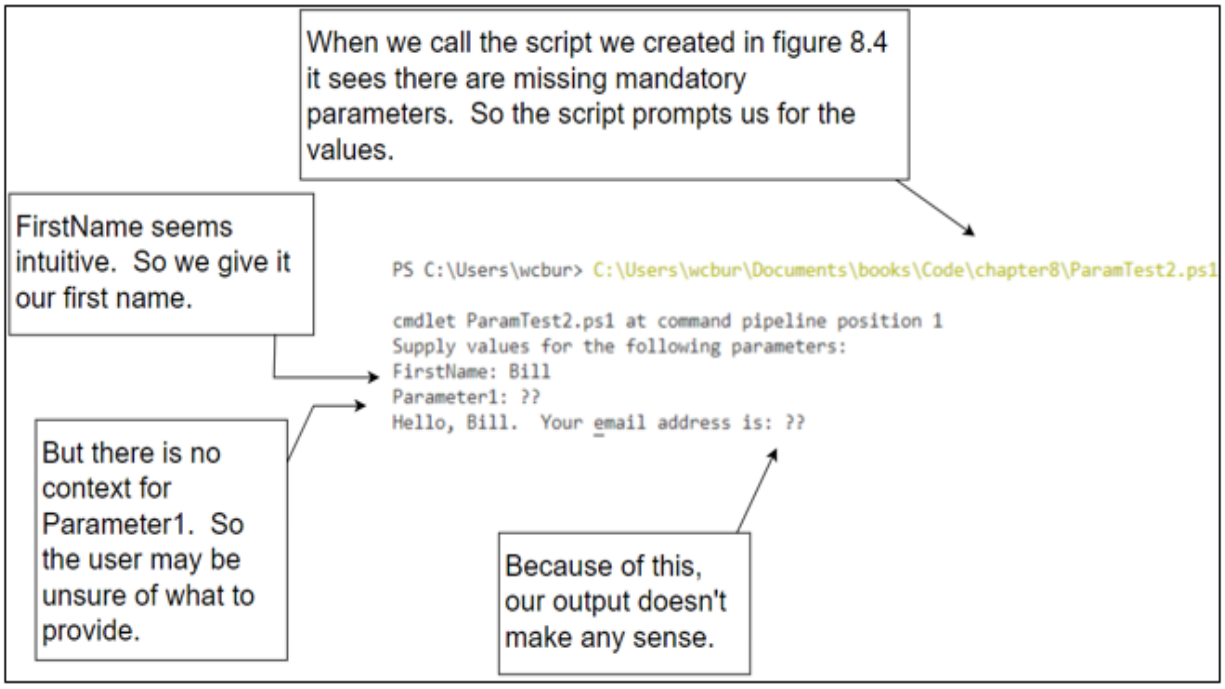


Figure 8.5 Prompting for missing Mandatory Parameters.

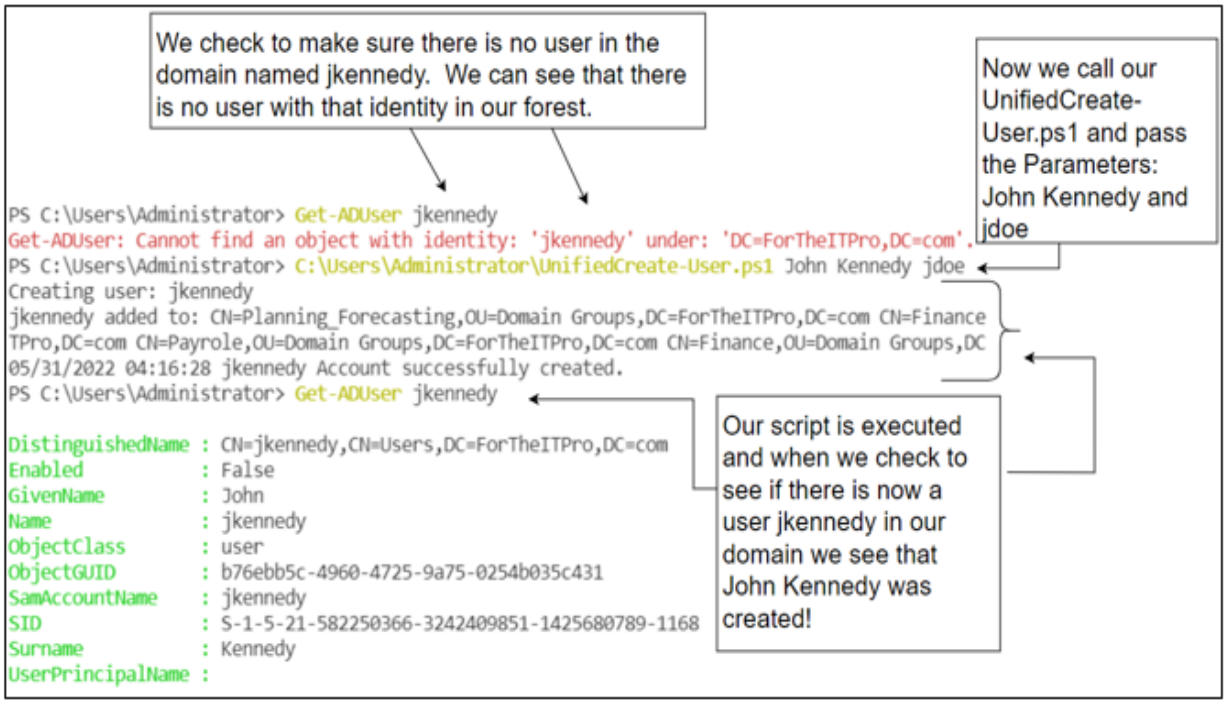


8.2.2 How we use Parameters in our Script

We specify three parameters for our UnifiedCreate-User script. These parameters are named intuitively and flagged as mandatory. This allows for our script to be called and have the required parameters included; without having to have any user interaction to capture or create these variables. We will see in the next chapter how we can utilize this to make hundred or even thousands of unique user accounts because of these parameters within this script.

When the script is called, if any of these parameters are omitted the script will prompt the user for these missing values. So there is no foreknowledge of the parameters required.

Figure 8.6 Running UnifiedCreate-User script with Parameters



8.3 PSSession

If you were to try to run the Enable-Mailbox cmdlet on your local computer you'd most likely receive an error message stating that the term "Enable-Mailbox" is not recognized as a cmdlet, function, script file or operable program see Figure 8.7.

Figure 8.7 Error generated by not having Exchange module loaded.

This essentially means that PowerShell is unable to recognize the term Enable-Mailbox because the Exchange module is not loaded. The easiest and most efficient way to gain access to the Exchange related cmdlets is to run the cmdlet on an Exchange server in your domain. When Exchange is installed on a server, by default, all of the PowerShell cmdlets are installed too.

However, we're trying to make a unified user creation solution, we don't want to run part of it on one computer, then stop, login to an Exchange server, and run a different script. There must be an easier way.

There is! A PSSession is a persistent connection we can establish to a remote computer. If you remember in chapter 4, we discussed that local variables were stored in our local session. By establishing a PSSession to a remote computer we're able to run commands that share data including those variables, or even the contents of a function. This allows us to create a connection once to a remote computer, then execute several commands.

8.3.1 New-PSSession

In order for us to create a persistent connection to a remote computer we first need to use the New-PSSession cmdlet.

Persistent vs Temporary Connections

In the next chapter we'll discuss remoting into computers. When we do this, we often use the Invoke-Command to run a specified command or cmdlet on a remote computer. This utilizes an unmanaged PSSession known as a Temporary session. PowerShell creates the session, executes the command and terminates the session all without any input from us.

These temporary sessions are useful for executing commands on endpoint computers. But any items created within the session will be lost when the session ends (typically as soon as the command is executed on the remote computer.)

A persistent connection, however, is just that, persistent. The connection is defined and managed, it is established when we want it to be and it ends

when we disconnect the session, or after a timeout period. During this time, objects created in either PSSession, your default session, or the remote session can be accessed and modified—such as the contents of variables or arrays, the contents of a function, or definition of an alias.

Typically, the syntax for creating this connection would be:

```
New-PSSession -ComputerName RemotePC
```

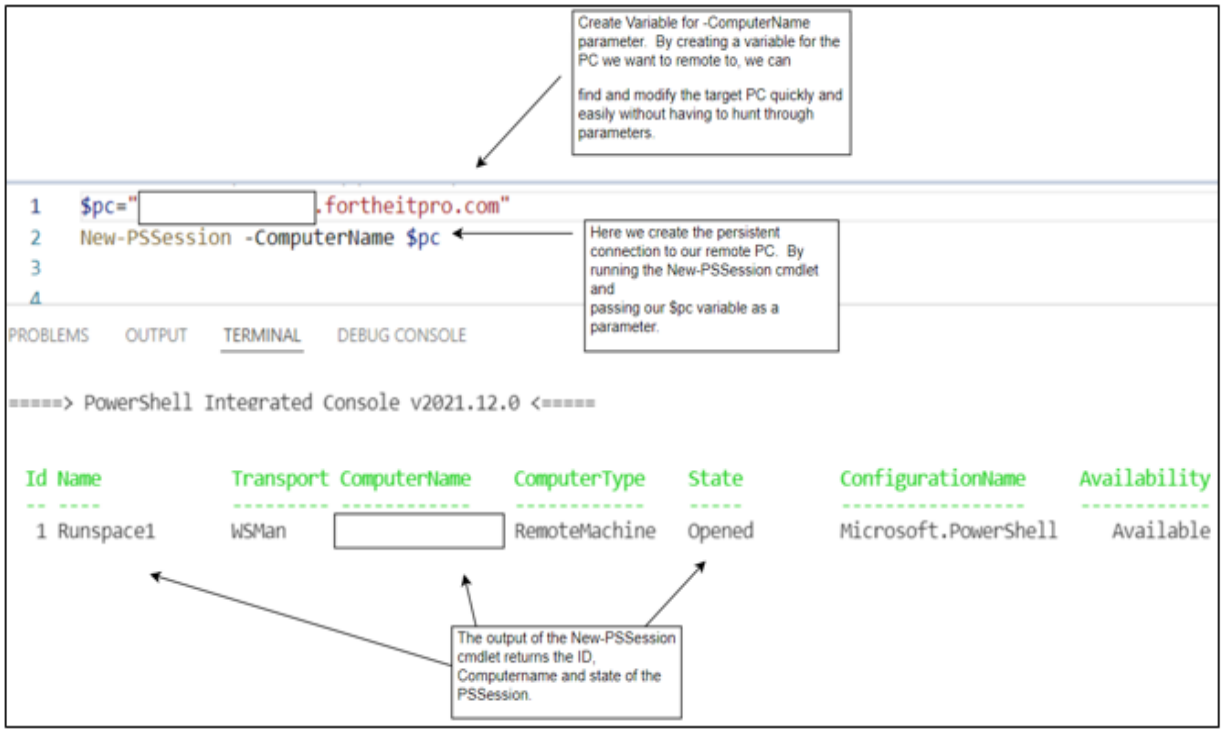
Note

It's possible to create multiple PSSessions in a single line by separating each PC by a coma.

```
New-PSSession -ComputerName RemotePC1, RemotePC2, RemotePC3
```

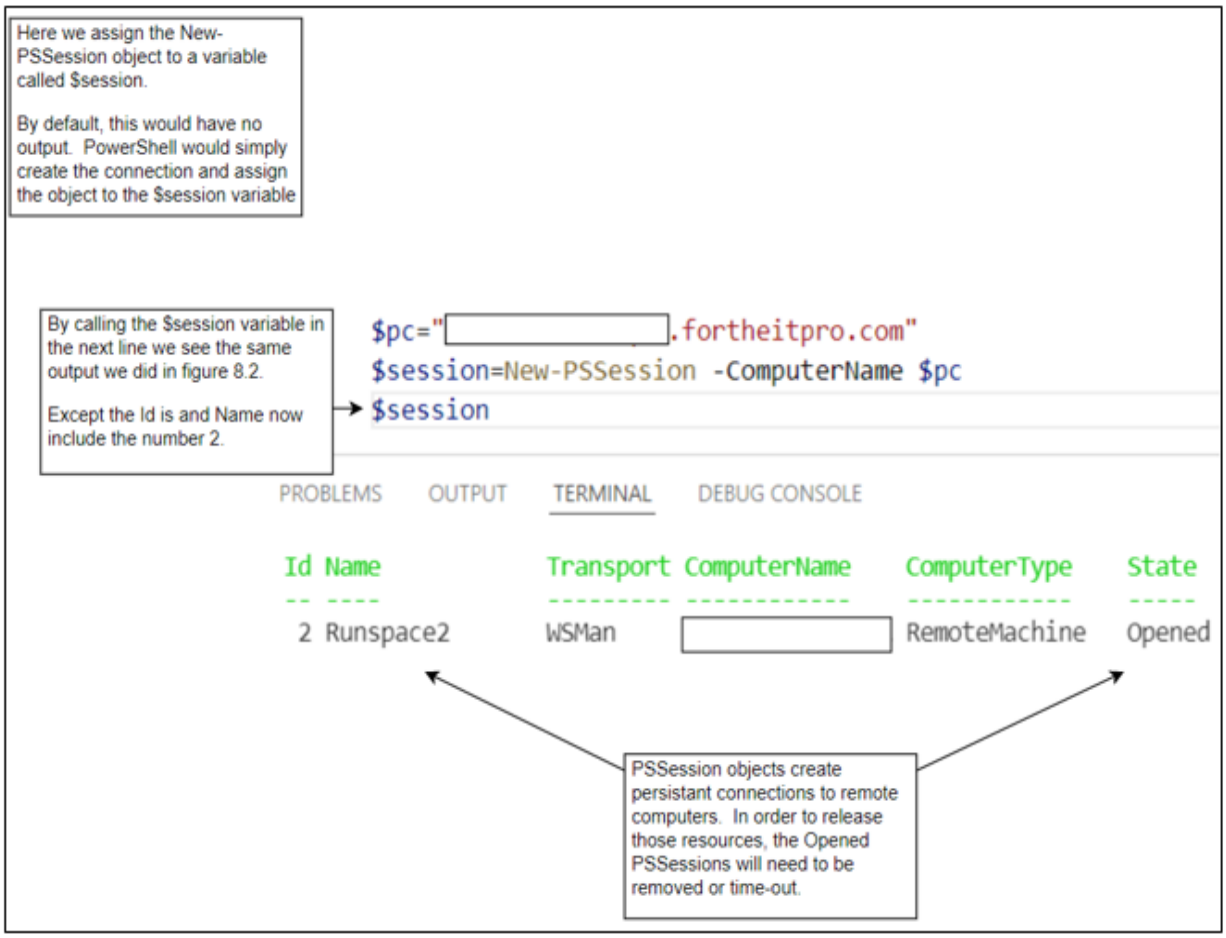
This code would create persistent connections to each of these three remote computers.

Figure 8.8 New-PSSession cmdlet example.



This object can be saved to a variable for later use in managing the connection, reconnecting, or removing the session without having to look up the session information.

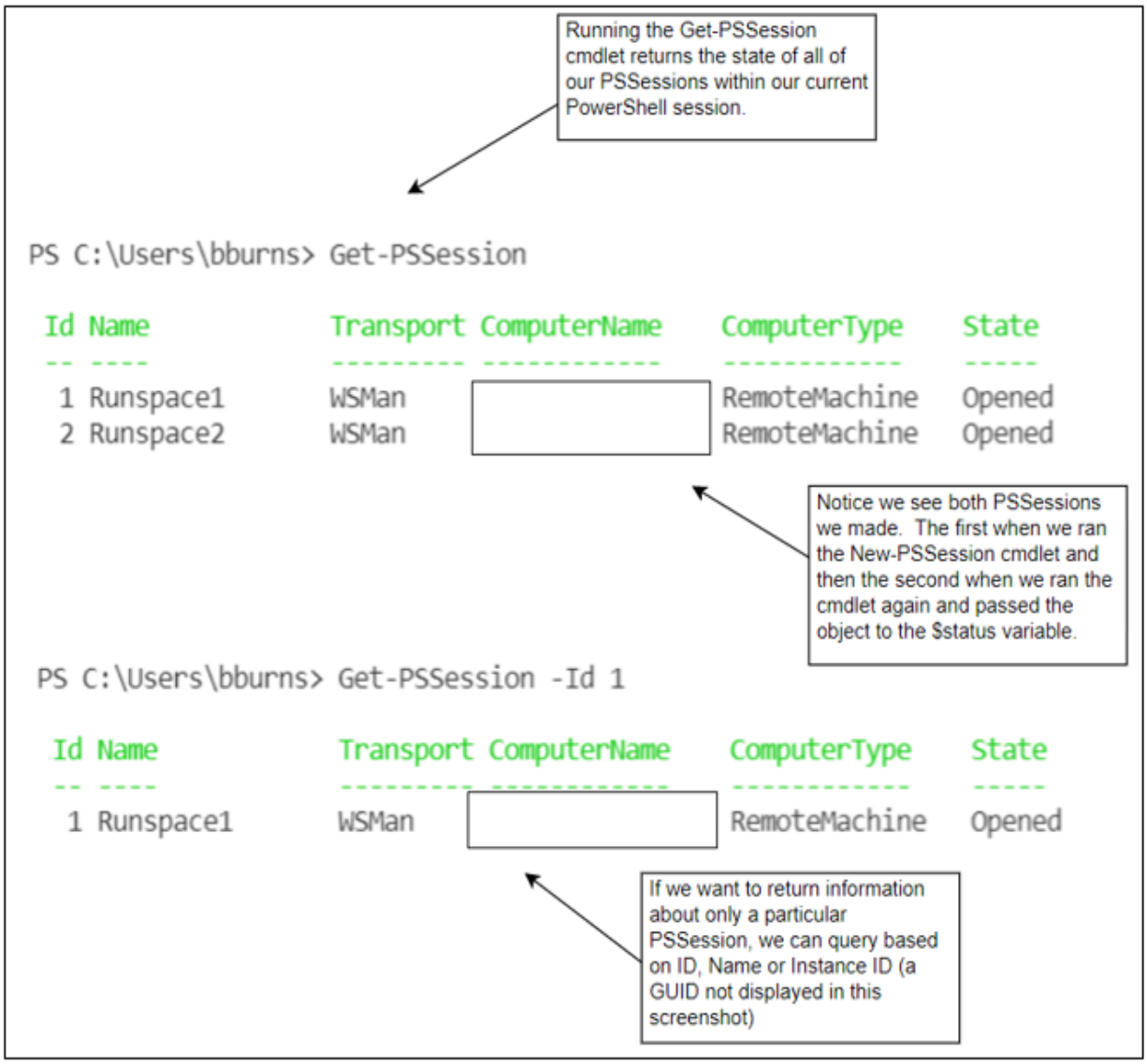
Figure 8.9 Saving PSSession object to variable example.



8.3.2 Get-PSSession

We can see the status and information of each of our PSSessions by running the Get-PSSession cmdlet.

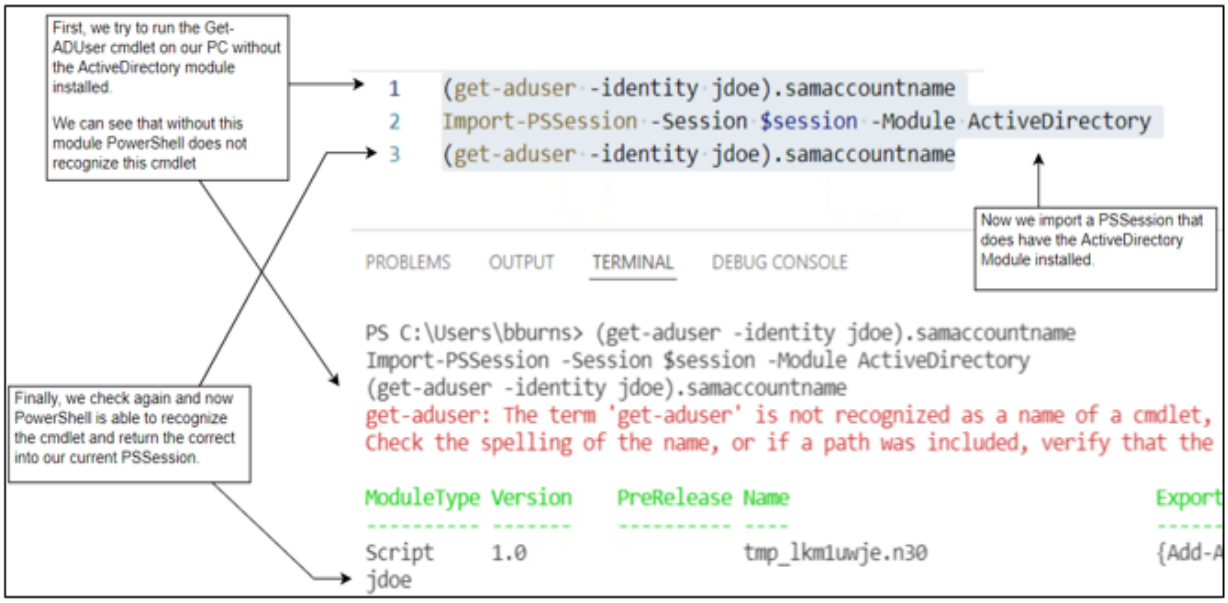
Figure 8.10 Get-PSSession example.



8.3.3 Import-PSSession

Now that we have established a connection to a remote computer, we want to make sure we can import the cmdlets, aliases and functions from our remote computer into our current session. To do this, we utilize the Import-PSSession cmdlet.

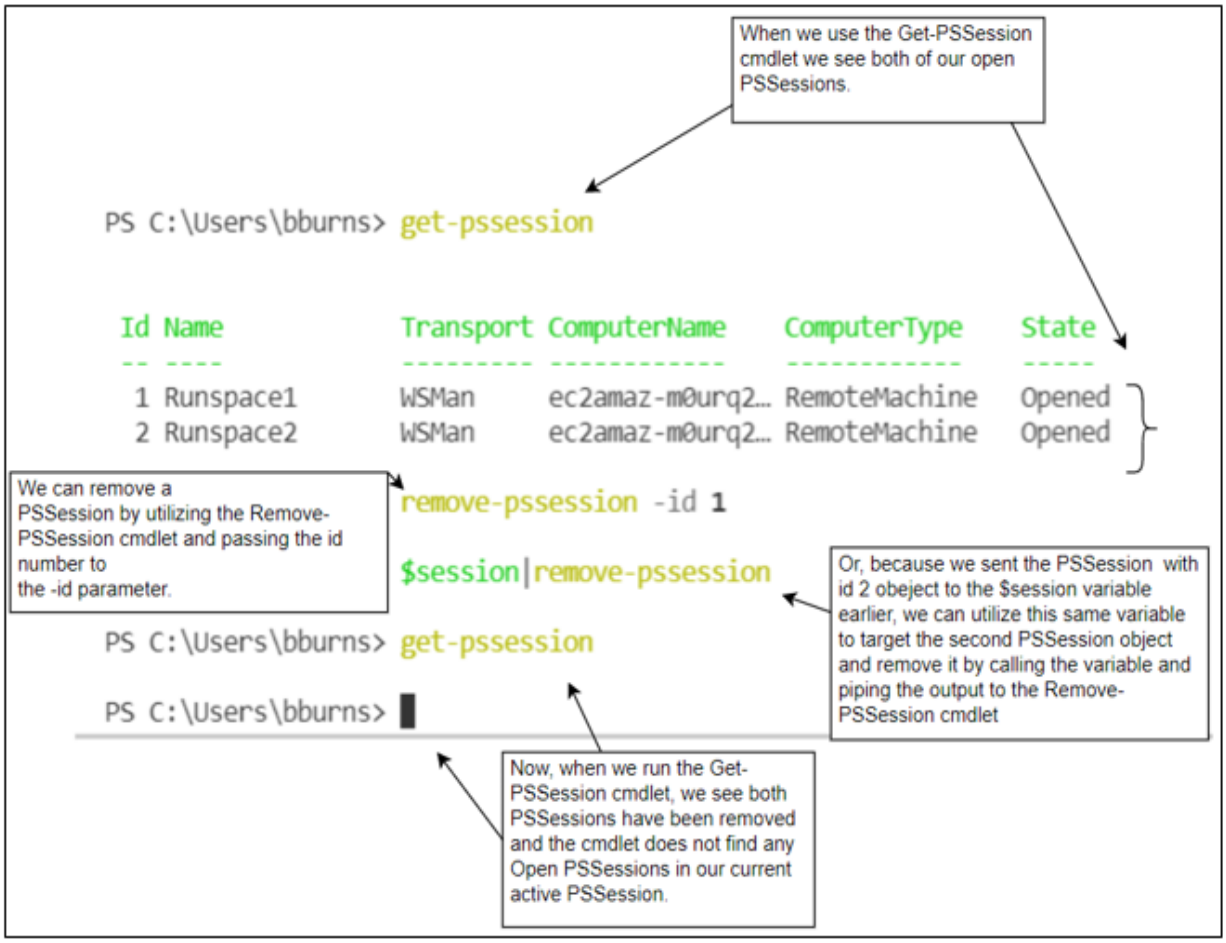
Figure 8.11 Utilizing Import-PSSession to use remote ActiveDirectory cmdlets.



8.3.4 Remove-PSSession

Remember, these are persistent remote connections. This means that they will stay active on both your computer and the endpoint computer until you close them, or they time out. While computer resources are not nearly as finite as they were in decades past a number of open remote PSSessions are a resource drain on your environment. It's best practice to terminate those remote PSSessions when they are no longer needed, freeing up resources on both ends.

Figure 8.12 Removing PSSesssions



8.3.5 How we use PSSessions in our Script

Now that we see how PSSessions can be utilized to leverage PowerShell modules across remote computers seamlessly in our PowerShell Session, let's examine how we utilize this in our Tiny PowerShell project code.

Note

Because books have a finite width, I have spanned several lines by breaking up what could be a single line of code into multiple lines of code. As notated in the README.md and in Chapter 3 [figure 3.1], in PowerShell 7 you can natively span lines at any pipe (|).

But due to width concerns I have broken the lines not at pipes (|), but instead at cmdlet parameters. We can use the backtick (`) to tell PowerShell that our

command naturally follows on the next line.

```
Get-ADuser -Identity jdoe -Properties MemberOf
```

Could be written and executed in PowerShell as:

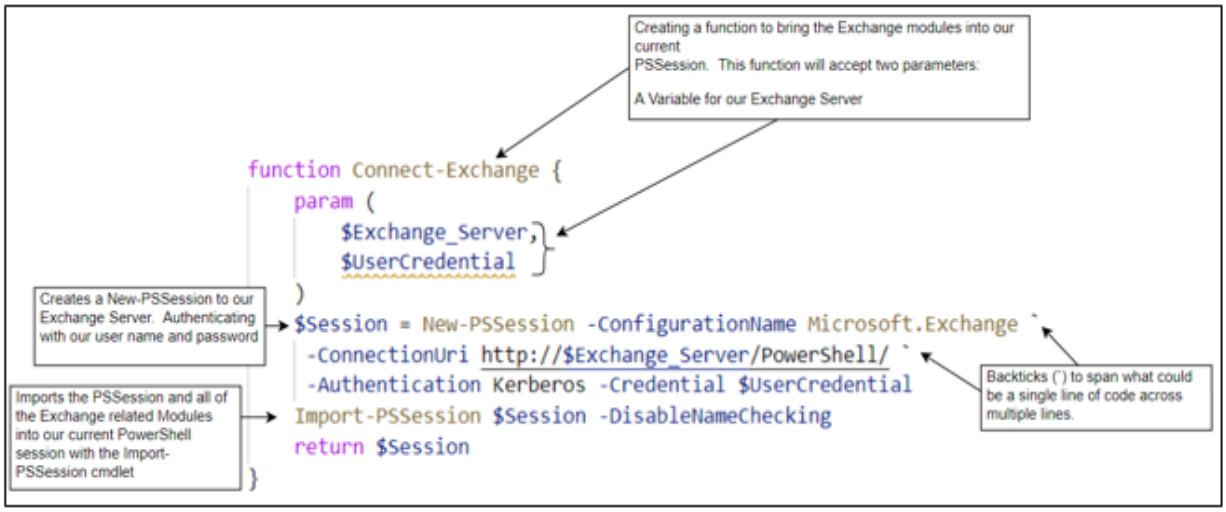
```
Get-Aduser `
-Identity jdoe `
-Properties MemberOf
```

We will make use of these backticks (`) to span lines in the next section.

Our project code will:

- Start with a function Connect-Exchange we can use to connect to our Exchange server and import the Exchange module into our current PSSession.
- The function will accept two parameters, $ExchangeServer and $UserCredential. Both of these values will be passed to the function when it is called.
- A variable called $Session will be created and will store the object information for our New-PSSession that targets our Exchange server using the $ExchangeServer variable as the connection Uri and the $UserCredential as the authentication password to establish this persistent remote session.
- The script will then import all of the Exchange modules into our active PSSession utilizing the Import-PSSession cmdlet.
- Finally, it will return the object holding the information for our PSSession to our main script. This will be used, eventually, to close the connection without having to utilize any Get-PSSession information.

Figure 8.13 Tiny PowerShell project code: Connect-Exchange function using PSSessions.



Note

Up until now, when we have used conditional logic, like an If statement, we have used a comparison operator to evaluate the statement

```
($answer -like "y*")
```

However, the comparison operator is not always required. Sometimes we don't know what the value of the data will be. Sometimes all we want to know is if there is something stored within the variable or not.

If for example we asked our user for their name, we don't have any idea what name they might enter. We could write 26 -like operators

```
(($name -like "a*") -or ($name -like "b*") -or Etc…))
```

But if we simply use the statement:

```
($name)
```

This would do the same thing. It would look at the value of the variable $name and if anything was in there it would return a true value. If, however, the value was empty (or null, covered later in this chapter) the statement would return false.

8.4 If not

When you are using conditional logic, often you want to make sure that something is not equal to a value instead of testing for a specific value. If you were patching computers in your domain and it required a reboot, you would want to make sure you avoided patching your servers, so your critical systems didn't go down unexpectedly. You might have a handful of servers, but you likely have hundreds, perhaps thousands of computer systems.

Wouldn't it be nice to have a way to filter on the thing you don't want and not the thousands of things you do want?

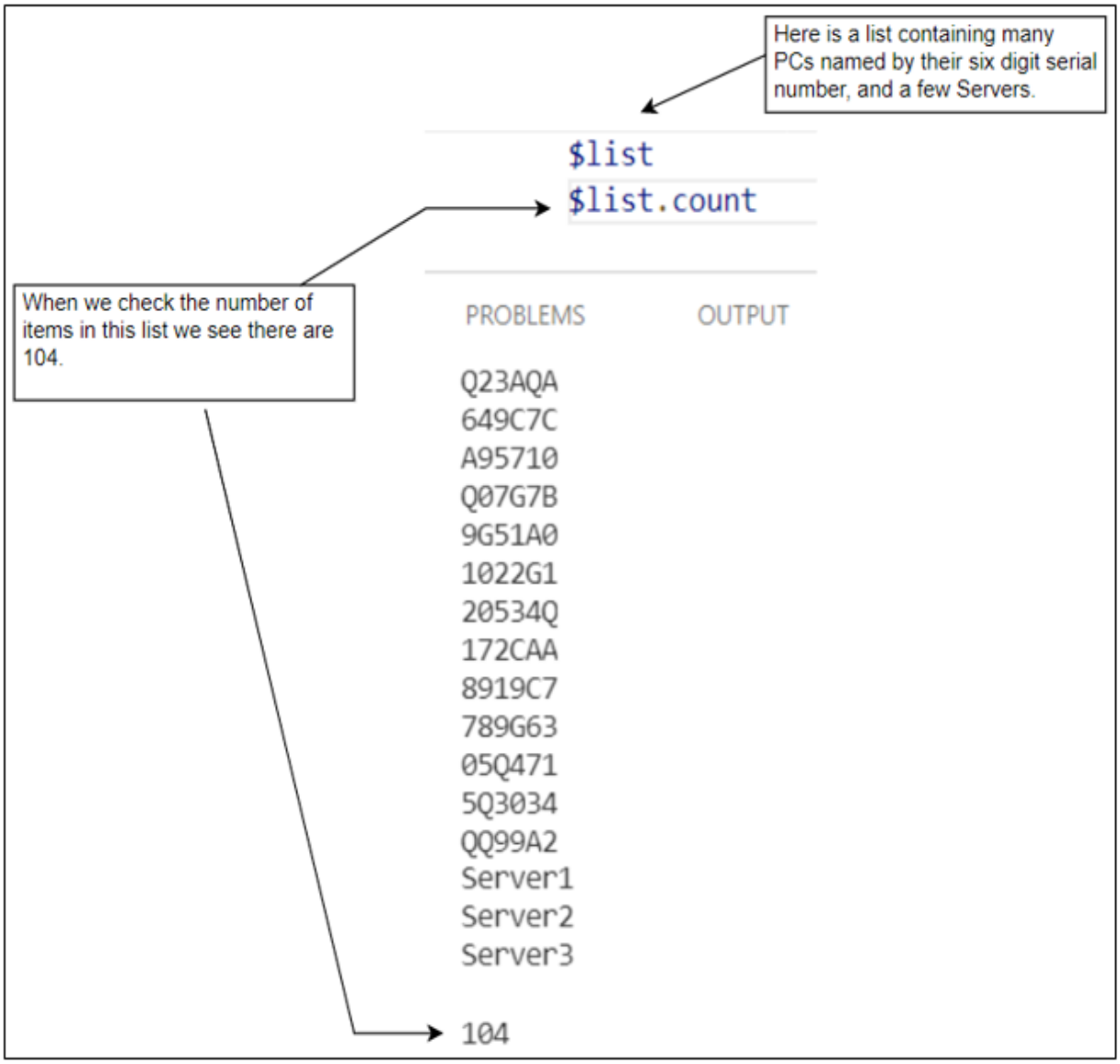
Fortunately, PowerShell makes this easy. First you can simply modify the comparison operator. Many of them have a "not" value:

Table 8.1 Comparison Operators and Not

Equal	-eq	Not Equal	-ne
Like	-like	Not Like	-notlike
Not	-not		

In the example below there are 104 listed computers. The PCs are named by their six-digit serial number. Servers are called Server1, Server2, and Server3. Instead of filtering for all of the possible combinations of six-digit serial numbers, it would be much easier to simply grab everything that wasn't a server.

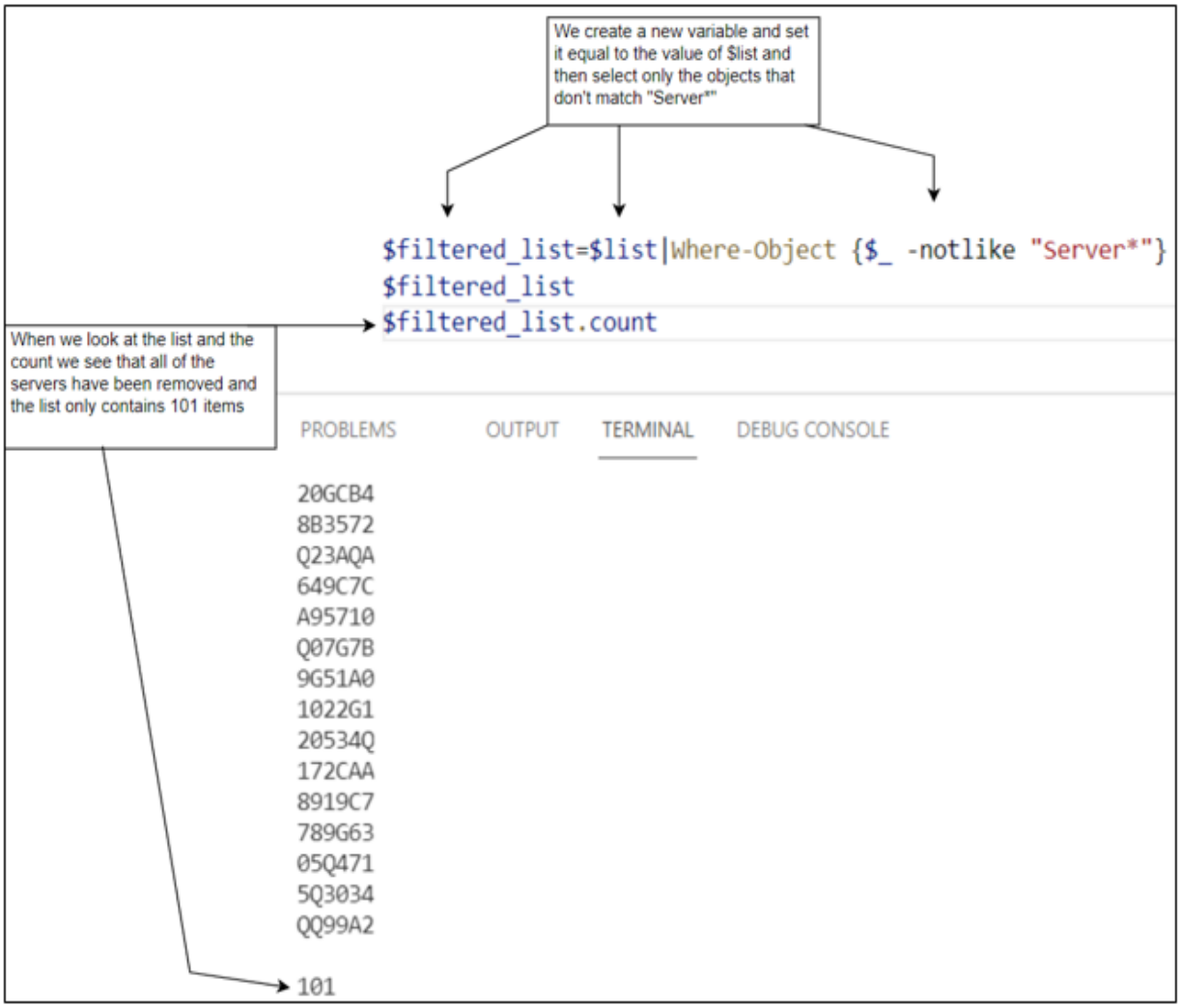
Figure 8.14 Non-filtered list of Computers and Servers.



To select non-servers, we will first create a variable called $filtered_list and set the value of this variable to our $list variable. We will then select only the objects in the list that do not start with the word "Server" by utilizing the -notlike comparison operator.

When we check we see that all three servers have been removed and the count is now 101, meaning that all the rest of the systems in the list are unchanged.

Figure 8.15 Filtered list using the -notlike operator.



8.4.1 How we use If Not in our Script

In our Tiny PowerShell script, we define the variable that will contain the FQDN for our Exchange server. This is best done as a variable, because it's possible that you may upgrade or move your Exchange server well before this script stops being useful in your environment. If this value was hard coded in your script you would need to find every instance of this value in your script and change it. By capturing it as a variable, you will only need to change this value in one logical place if it ever changes.

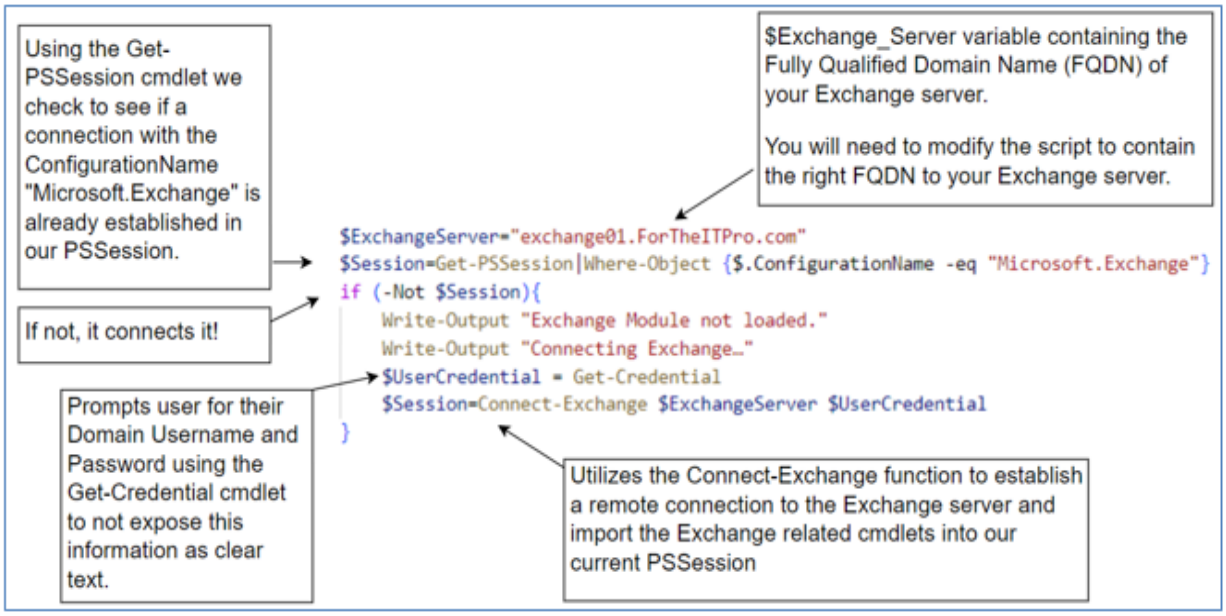
Next our script will use the If -Not logic to see if the PSSession has already been connected. It is a drain on our networking and computing resources to establish multiple remote sessions. Utilizing the If -Not logic we can

establish this connection to our Exchange server only if a connection is not already present. This will save us time and resources when executing this script.

We will use the Get-PSSession command and check to see if a PSSession exists in our current PSSession named "Microsoft.Exchange". It is slightly less efficient to filtering but doing so prevents an error that would otherwise break our script should the PSSession not exist. We will address how to avoid these errors in future chapters without filtering.

If the PSSession named "Microsoft.Exchange" is not connected to our current PSSession, we connect it.

Figure 8.16 Using if -not in our script

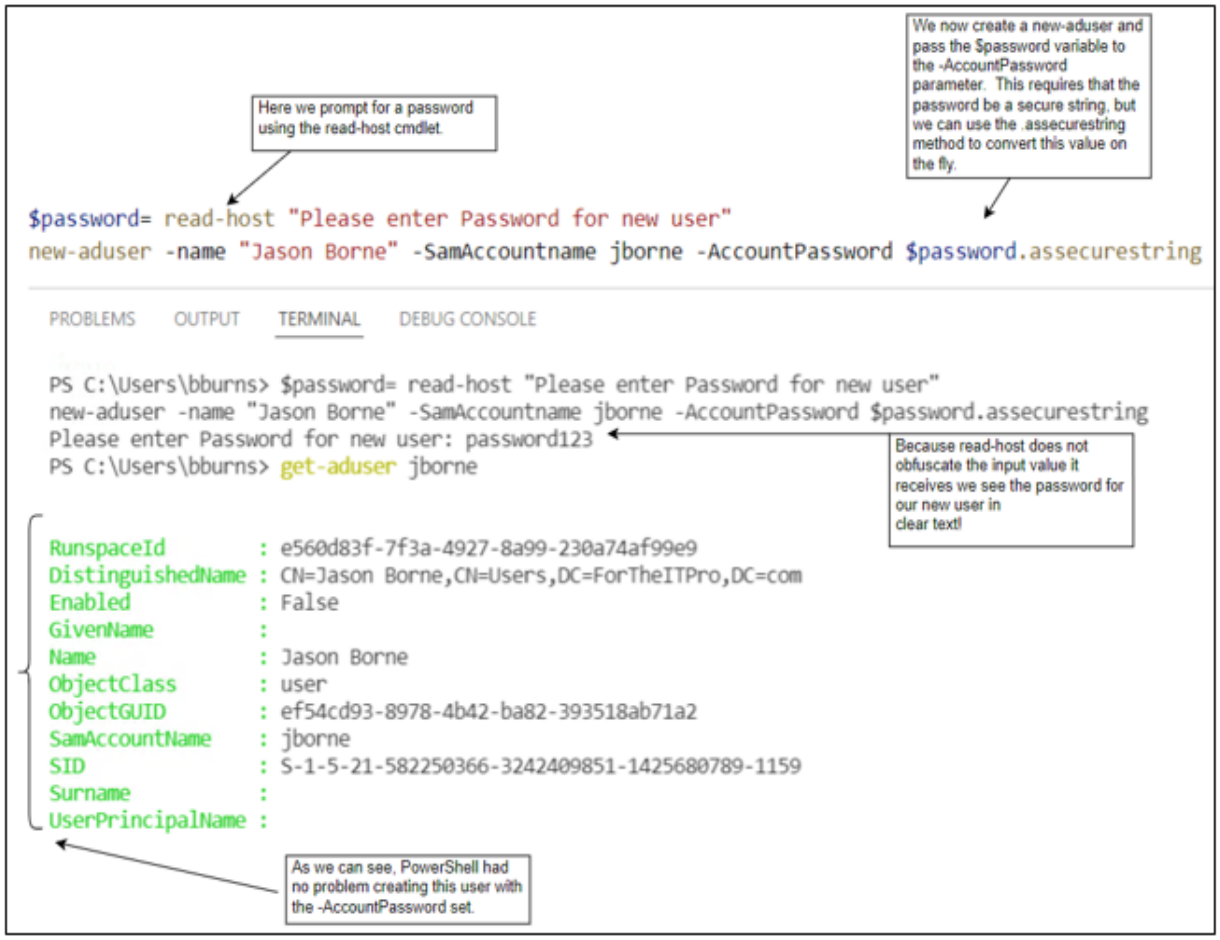


8.5 Get-Credential

Very often, you will need to provide a system your password to authenticate to the resources you are trying to access. We have learned so far how to prompt our user for input and capture that input as a variable. Most of the time this is fine, but when dealing with sensitive data we will likely want to re-consider this practice.

Below is an example of how using the Read-Host cmdlet would potentially expose our user password to a number of people.

Figure 8.17 Exposing Passwords in clear text with read-host cmdlet



What is a Secure String?

When you create a secure string, you create an encrypted PowerShell object known as a credential object. This credential can be converted from a text with the ConvertTo-SecureString cmdlet or, as we just saw, the AsSecureString method available to a PowerShell string object.

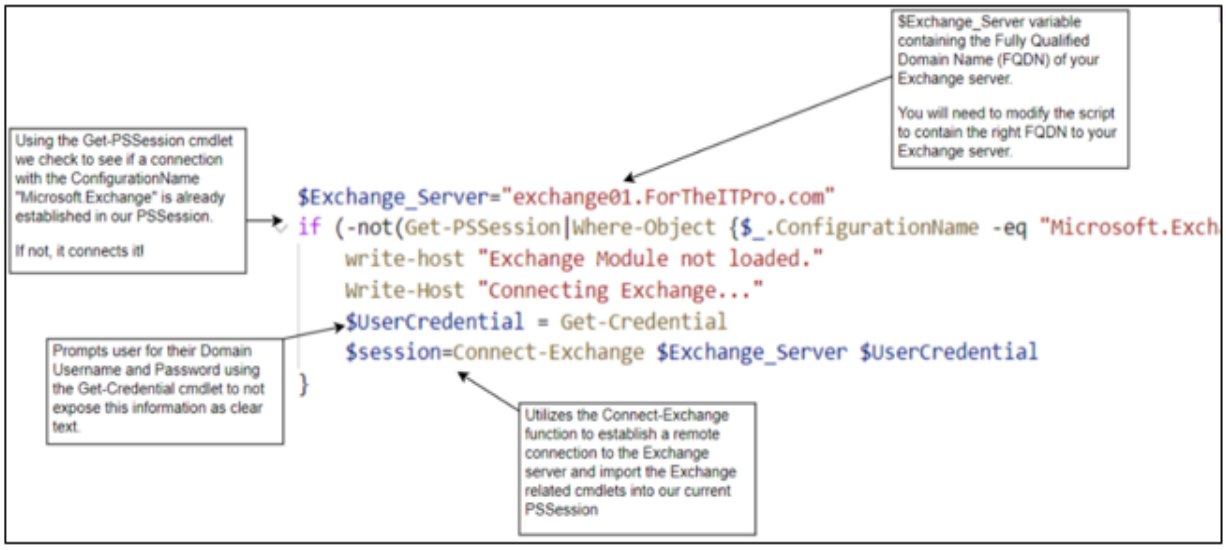
When PowerShell creates a credential object it encodes the object with a private key on your computer. When dealing with credentials, many cmdlets only accept PowerShell Credential objects, like the New-ADUser cmdlet above.

Wouldn't it be better if we were to capture a user credential while masking keyboard input? That is precisely what the Get-Credential cmdlet does.

Figure 8.18 Using Get-Credential to prevent exposing passwords in clear text.

8.5.1 How we use Get-Credential in our Script

Figure 8.19 Get-Credential used to establish Remote PowerShell Exchange Server Session.



To create a new email address, we need the capabilities provided to us by the PSSession on the Exchange server. If the PSSession named "Microsoft.Exchange" is not connected to our current PSSession, we prompt our user for their username and password with the Get-Credential cmdlet. This cmdlet prevents the transmission of this information in clear text.

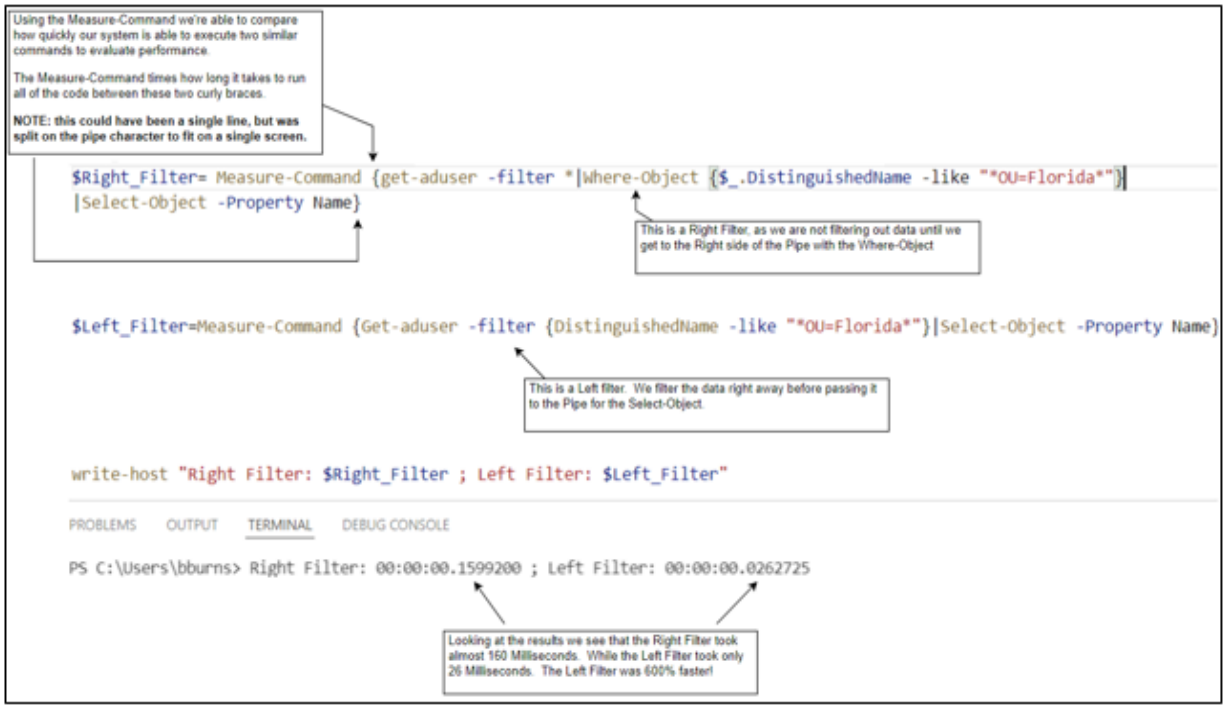
8.6 Filtering performance

In the PowerShell community you'll likely hear the term "Filter Left" from time to time. This means that it's usually best practice to filter early in your pipeline Because pipelines are read and executed from left to right, the sooner you can filter for what you're looking for the less data you'll have to deal with in later steps of the pipeline. This means that your script will run faster as it's processing less data.

8.6.1 Measure-Command

One way to determine which process takes more computing time is to use the Measure-Command cmdlet. Much like it sounds it simply returns the total time that the command took to run. The syntax is to simply enclose the command or groups of commands you wish to run within curly braces. By default, the Measure-Command cmdlet returns a TimeSpan object. The format of the default output is: Days:Hours:Minutes:Seconds:Milliseconds.

Figure 8.20 Example of Left and Right filter with Measure-Command cmdlet



Obviously, best practice is to use the better performing left filter. That being said, in our Tiny PowerShell Project code we do not use this practice when detecting for the existence of our Exchange connection.

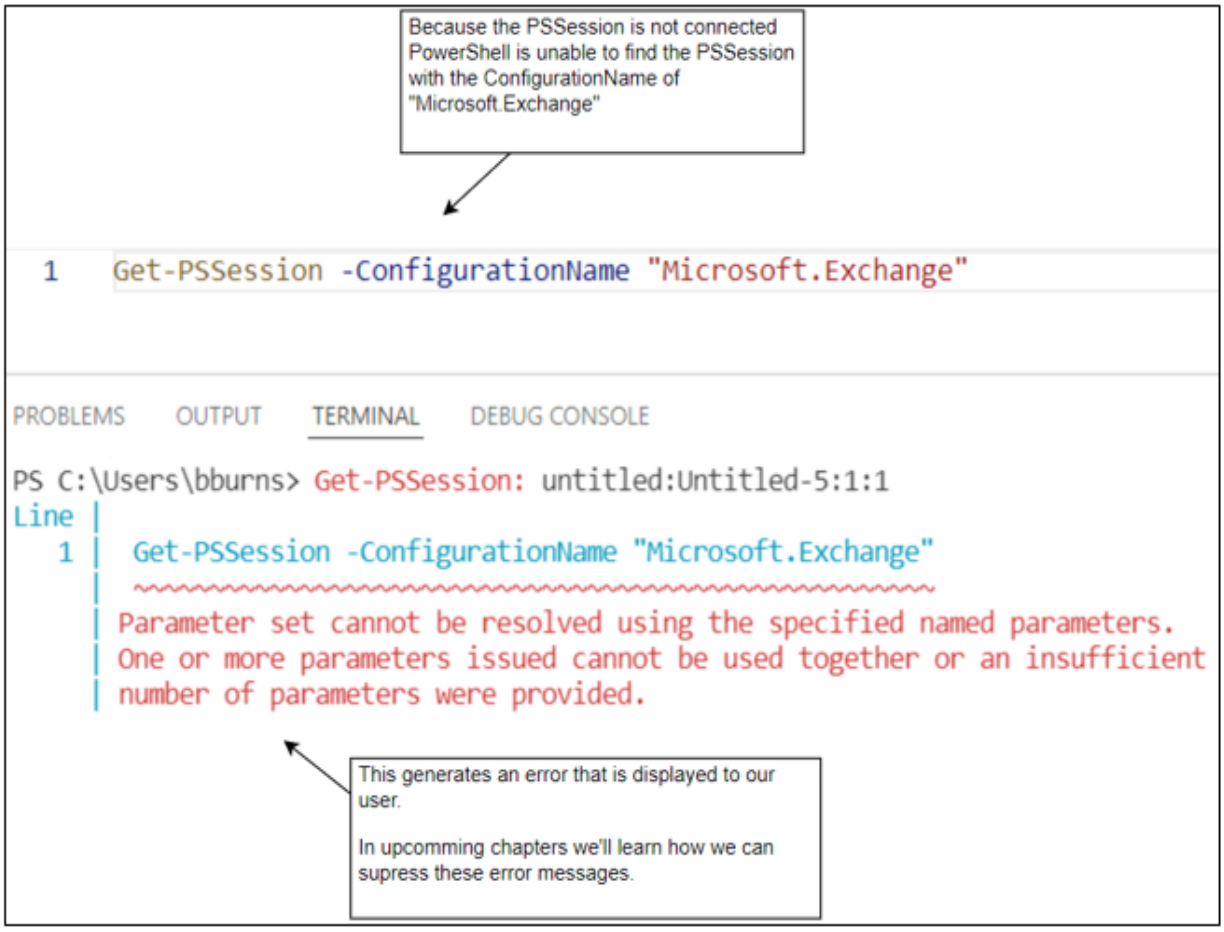
```
if (-Not(Get-PSSession|Where-Object {$_.ConfigurationName -eq
"Microsoft.Exchange"})){
```

The code could be written to execute faster by filtering earlier in the statement.

```
if (-Not(Get-PSSession -ConfigurationName "Microsoft.Exchange"))
{
```

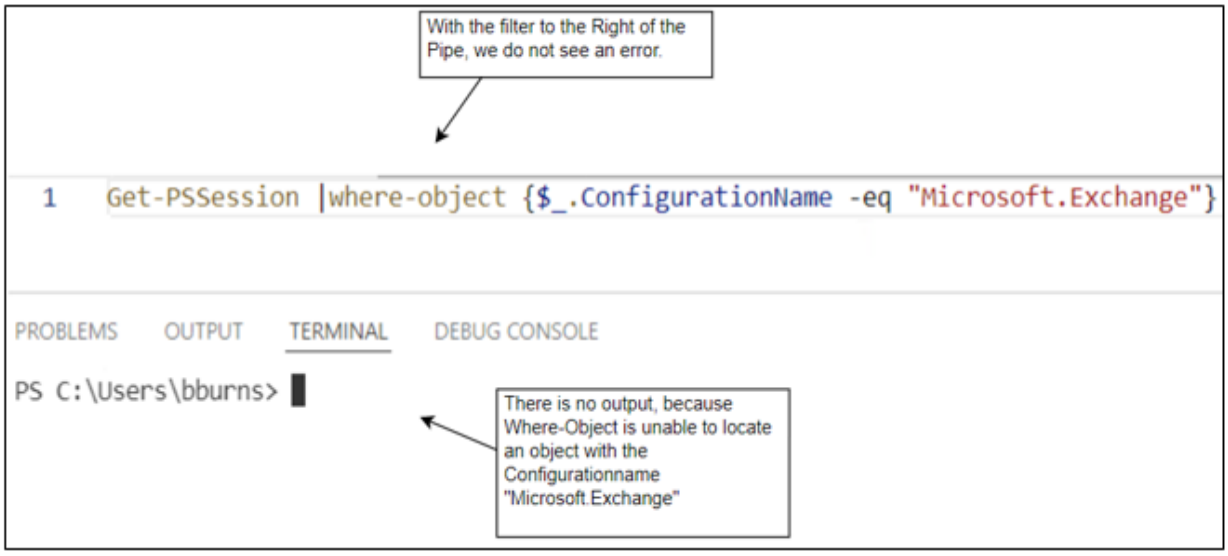
Doing this, however, when the PSSession is not connected throws an error seen below.

Figure 8.21 Error generated by filtering left



When we use the code as written in our Tiny PowerShell project with a filter to the right of the pipe, we see there is no error.

Figure 8.22 No error when filtering right.



Note

There is a way to suppress this error in the left filter above. We will explore the -ErrorAction parameter in later chapters of this book.

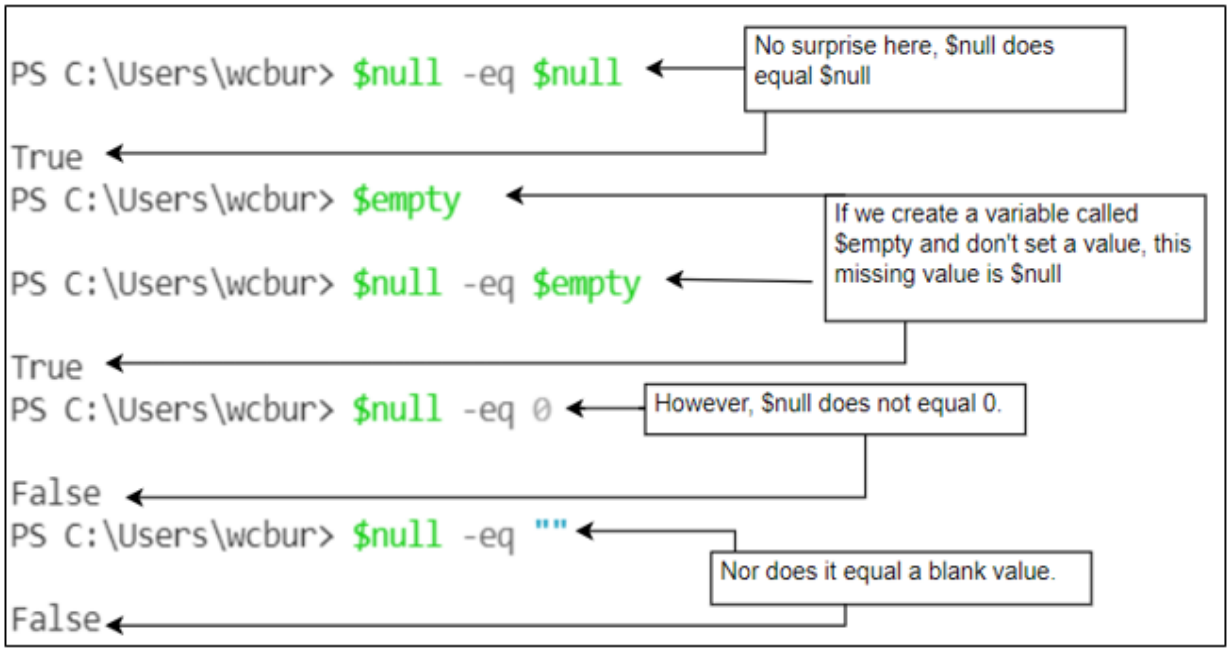
8.6.2 Null values

Null is a programming concept that can take a little bit to master. In PowerShell, think of Null as an unknown or empty value. Null is not the same thing as an empty string or zero. Null is an absence of a value.

If you create a variable but do not assign it a value, the value of that variable is $null. However, if you create a variable and assign it even an empty string, the value is no longer absent; it is empty; so, this is no longer Null.

Below we'll see several examples of Null values and non-Null values.

Figure 8.23 Examples of $null.



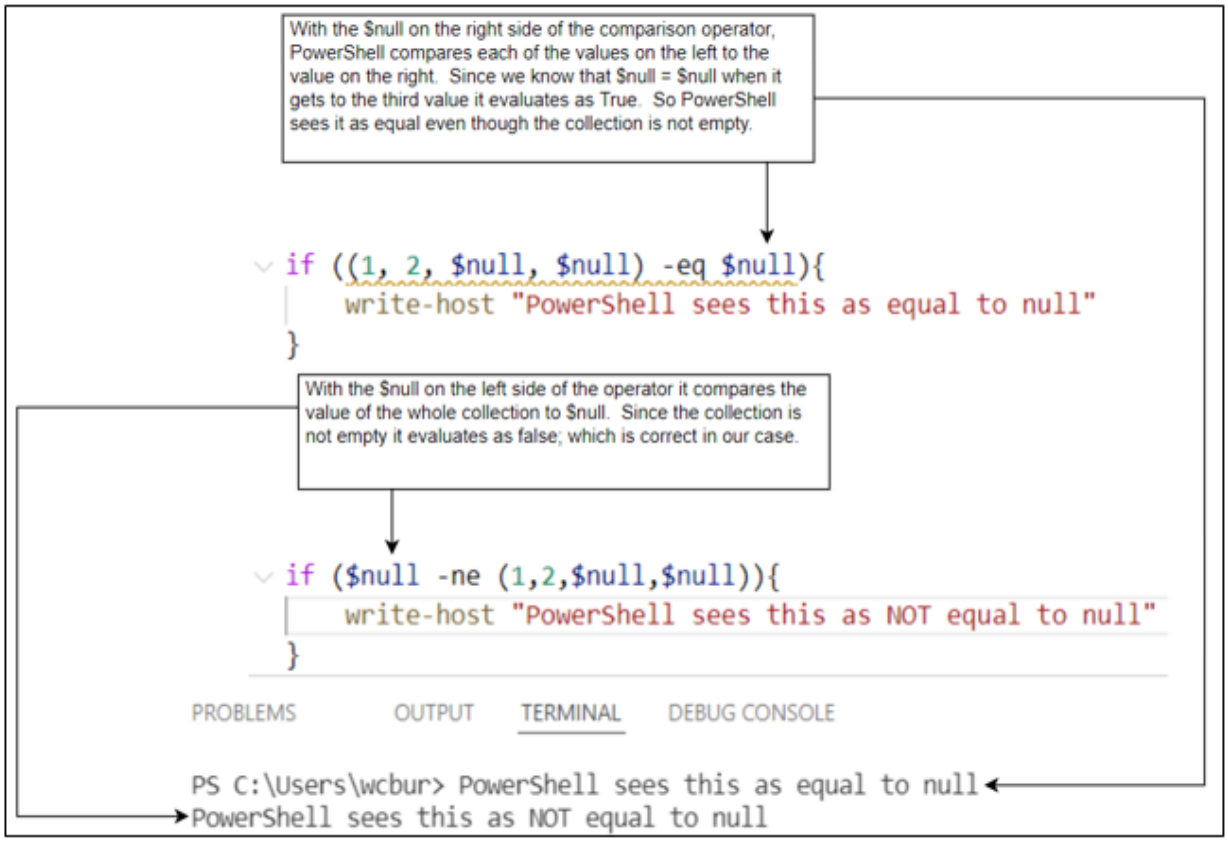
8.6.3 Filtering Null Values

If you notice, in the example above, the $null was always in the left of the comparison operator. This was intentional. Unlike every other example where we have used the comparison operator to date, when we check the value of $null we typically place the $null in the to the left of the operator.

This is because $null is what's known as a scalar value. What this essentially means is that the value of $null is a singular value. When you place a scalar value in the left-hand side of the comparison operator it compares the whole object on the right-hand side to the value on the left. Meaning that the whole value on the left must be null to be evaluated as null.

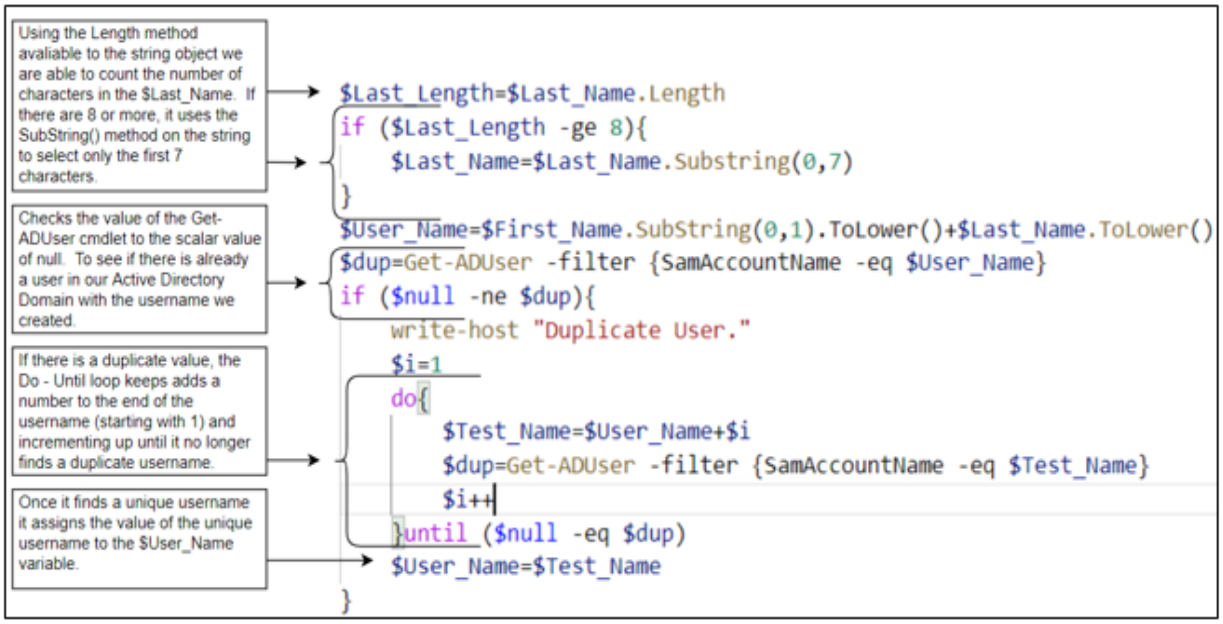
If the value on the left-hand side is a collection, PowerShell compares every value in that collection to the value on the right-hand side. So, if one of these values on the left is null it will match the null value on the right. The easiest way to understand this is to see the figure below.

Figure 8.24 Left hand vs right hand $null example



8.6.4 How we use NULL values in our Script

Figure 8.25 Using Null values to check for duplicates.



When creating users with our script we check to see if the user exists checking to make sure that the value of our SamAccountName does not exist on our domain. We do this by checking to make sure that the value returned is NULL.

If the value does come back as one already in use on our domain we use a do-while loop checking each iteration of SamAccountNames until one evaluates as NULL, so we can safely assign this SamAccountName to our new user without fear of duplicating a username.

8.7 Summary

- The New-PSSession cmdlet allows you to create a persistent connection to a remote computer.
- The Get-PSSession cmdlet will show you a list of all of the remote PSSessions connected to your current PSSession.
- Import-PSSession allows you to import cmdlets, aliases and functions from a remote PSSession you are connected to directly into your current PSSession.
- Because persistent remote connections require both computing and networking resources, it's best practice to remove PSSessions

connected to your current PowerShell Session when you no longer need access to these resources.

- You can close existing PSSessions with the Remove-PSSession cmdlet. You can target the PSSession name, group ID or even an object variable you may have stored.
- There are situations where it's more efficient to use conditional logic to search for something that may not be present, or filter for something you don't want, rather than list all of the things you do want. PowerShell allows for this with its many -not comparison operators: -not, -ne, -notlike, and -notmatch grant significant flexibility to your conditional logic.
- The Get-Credential cmdlet is a way to prompt a user for their username and password without exposing the input as clear text. If you are prompting a user for this information, you should always make sure to protect this Personal Identifiable Information (PII) at the highest level possible.
- PowerShell commands are read from top to bottom and left to right. Therefore, it is more efficient to filter as far to the left of your statement as possible.
- You can use the Measure-Command cmdlet to measure the length of time your command, or series of commands, takes to complete.
- Optimizing for performance will make all of your scripts run better and faster, saving you even more time.
- The null value in PowerShell represents an empty value or absence of data.
- It can be useful to check for the null value when using conditional logic. Be aware that the null value is a scalar value and should almost always be placed in the left-hand side of the conditional logic expression to make sure that the whole condition is indeed null.

9 Making LOTS of users!

This chapter covers:

- Additional Parameter options to collect credential objects and check for null values
- Building a Windows dialogue box to explore for a file using the GUI
- Using the While loop
- A new way to check for empty or null strings
- Reading data from a Comma Separated Value list (CSV)

In the last chapter, we took a big step forward in making our lives easier. We now have a script to create a user in our domain from start to finish with a single click and some simple values. This is great! But companies typically have some sort of onboarding procedure to bring batches of employees in at the same time. Thus, as an IT Professional, it's far more common that you are creating new users in batches.

We have a script that will create a new user, but do we want to run it over and over, perhaps dozens of times? Isn't there an easier way? There is!

This script will:

- Accept a path for a specific type of input file: a Comma Separated Value (CSV) list.
- Accept a username and prompt for a PSCredential to authenticate into our Exchange server.
- Check to ensure the username and password is not simply null. A username and password are required to authenticate to the PSSession on our Exchange server.
- Check to see if there is an established connection to the Exchange Server and if not, create one using the PSCredentials provided.
- Check to see if a path to a CSV file was provided; if not, create a Windows Dialog Box to navigate to the file using the Windows GUI.

- Read each CSV file line, pull values from specific headers, and assign them to variables within our script: FirstName, LastName, and CopyFromUserSamAccountName.
- Using these variables, the script functions as it did in the previous chapter, creating unique, valid user names, assigning email addresses, and copying relevant group membership for each user in the CSV.
- When the last user in the CSV is created the script removes the PSSession to the Exchange server.

9.1 Project Code

The code is below; like our other scripts, we'll cover it in detail. Here are a few things of special notice for this script.

```
param (
    [Parameter(Mandatory=$false)]    #A
    [string]$CSVLoc,
    [Parameter(Mandatory=$true)]

[System.Management.Automation.PSCredential]$ExchangePassSecure
#B

    )
function Connect-Exchange {
    param (
        $ExchangeServer,
        $ExchangePassSecure
    )
    $Session = New-PSSession -ConfigurationName
Microsoft.Exchange `
        -ConnectionUri http://$ExchangeServer/PowerShell/ `
        -Authentication Kerberos -Credential $ExchangePassSecure
    Import-PSSession $Session -DisableNameChecking
    Return $Session
}
#Variable for the FQDN to your Exchange Server
$ExchangeServer="exchange01.ForTheITPro.com"

$Session=Get-PSSession|Where-Object {$.ConfigurationName -eq
"Microsoft.Exchange"}
if (-Not $Session){
    $Session=Connect-Exchange $ExchangeServer
$ExchangePassSecure
}
```

```
Function Get-File(){    #C
[System.Reflection.Assembly]::LoadWithPartialName("System.window
s.forms")
    | Out-Null
    $FileDialogBox = New-Object
System.Windows.Forms.OpenFileDialog
    $FileDialogBox.filter = "csv (*.csv)| *.csv"
    $FileDialogBox.ShowDialog() | Out-Null
         return $FileDialogBox.filename
    }

while ([string]::IsNullOrEmpty($CSVLoc){    #D
    Write-Output "Navigate to the CSV file to create your
users:"
    $CSVLoc=Get-File
}

$CSVInfo=Import-Csv -Path $CSVLoc    #E

foreach ($user in $CSVInfo){
    $LastName=$user.LastName    #F
    $FirstName=$user.FirstName
    $CopyFromUserSamAccountName=$user.CopyFromUser

    $LastLength=$LastName.Length
    if ($LastLength -ge 8){
        $LastName=$LastName.Substring(0,7)
    }

$UserName=$FirstName.Substring(0,1).ToLower()+$LastName.ToLower(
)
    $dup=Get-ADUser -filter {SamAccountName -eq $UserName}
    if ($null -ne $dup){
        $i=1
        do{
            $TestName=$UserName+$i
            $dup=Get-ADUser -filter {SamAccountName -eq
$TestName}
            $i++
        }until ($null -eq $dup)
    $UserName=$TestName
        if ($UserName.Length -ge 8){
            $UserName=$UserName.Substring(0,8)
        }
    }

    $LogFile="C:\Logs\$UserName.log"
```

```
    Write-Output "Creating user: $UserName"|Tee-Object $LogFile
-Append
    New-ADUser -Name $UserName -GivenName $FirstName -Surname
$LastName `
        -DisplayName $UserName -SamAccountName $UserName -
Enabled $false
        |Tee-Object $LogFile -Append
    Start-Sleep 5
    $Groups=(Get-ADUser -Identity $CopyFromUserSamAccountName -
Properties MemberOf).MemberOf
    $Groups|Add-ADGroupMember -Members $UserName
    Write-Output "$UserName added to: $Groups"|Tee-Object
$LogFile -Append
    (Get-ADUser -Identity $UserName).SamAccountName|Enable-
Mailbox
    Write-Output "$(Get-Date) $UserName Account successfully
created."|Tee-Object $LogFile -Append
}
Remove-PSSession $Session    #G
```

9.2 More Parameters

We introduced script parameters in our last chapter. Now we are going to explore a few more useful options for this amazingly powerful capability of our scripting.

9.2.1 Mandatory

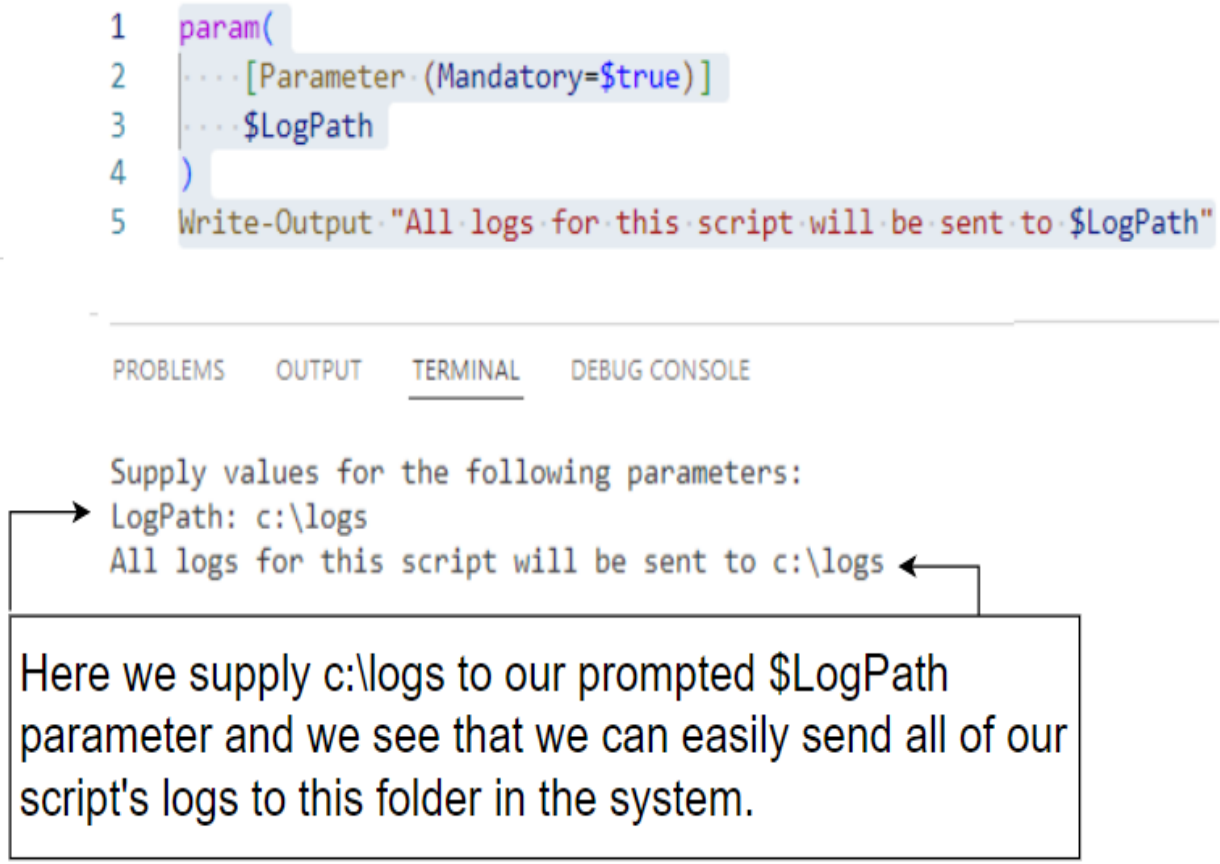
In our last chapter, we showed how we can use the [Parameter (Mandatory=$true)] parameter to prompt our user for a required parameter. But what if we don't want the parameter to be required, but instead optional? In that case, we can set Mandatory to $false, which allows users to run the script with or without providing a value for that parameter up front. In our project for this chapter, for instance, we allow the user to optionally enter a location for their CSV file when calling the script. This would save the user time since they would not have to navigate to the CSV file. But if they don't provide this parameter, the script still works.

Suppose we wanted to allow our users to specify a path for our logs to be stored. We may want to create this as a parameter that can be passed to our

script. That way, if our user has a folder they like to keep all of their logs in, it saves them the time of having to move all of these logs manually.

```
param(
    [Parameter (Mandatory=$true)]
    $LogPath
)
Write-Output "All logs for this script will be sent to $LogPath"
```

Figure 9.1 Example of mandatory parameters.



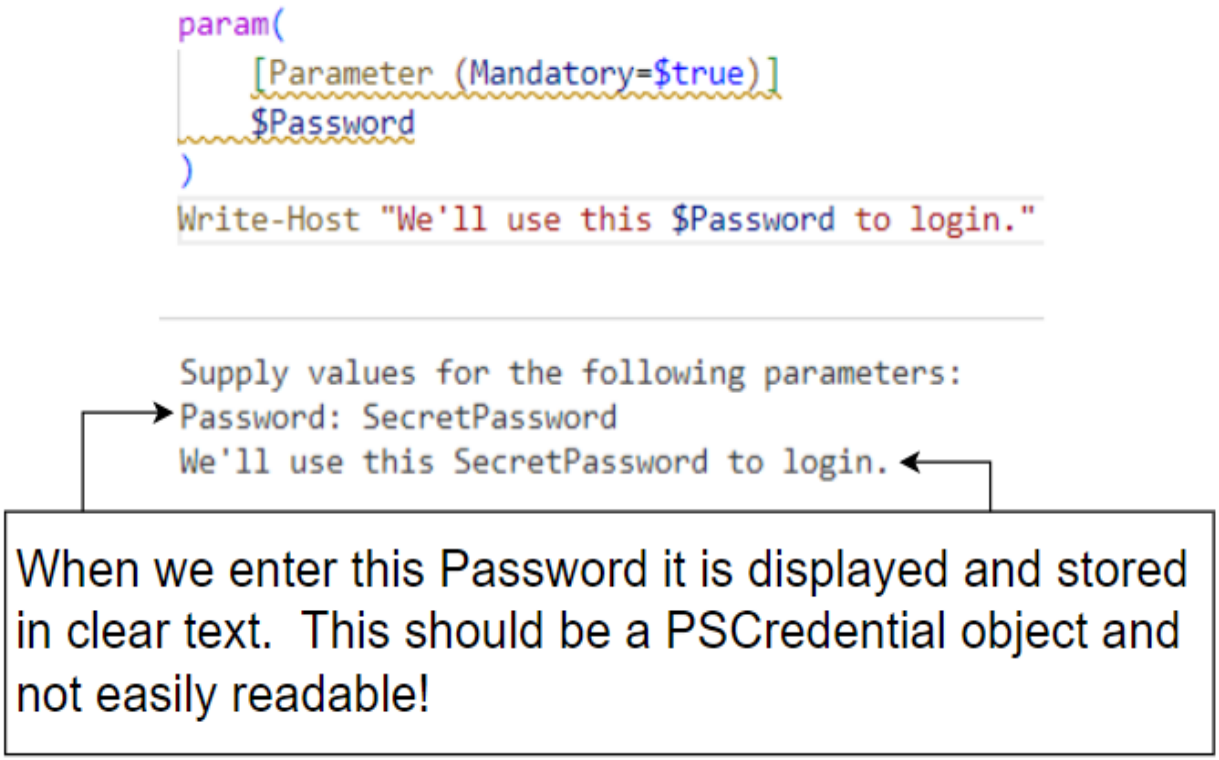
9.2.2 Working with PSCredentials

This works all right for paths, but what about passwords or credentials?

```
param(
    [Parameter (Mandatory=$true)]
    $Password
```

```
)
Write-Host "We'll use this $Password to login."
```

Figure 9.2 Passwords in clear text example



As you saw in the example simply storing the password as a string is an insecure and bad idea. Fortunately, with PowerShell, we can store our user name and password as a PSCredential object that never stores or transmits our sensitive login information in a cleartext format.

9.2.3 System.Management.Automation.PSCredential

We can solve this inconvenient problem by using the System.Management.Automation.PSCredential object type within the parameter block. This will prompt the user to supply a valid username and password that will become the PSCredential. A PSCredential is essentially a placeholder for usernames and passwords used as a safe and convenient way to handle this sensitive information.

```
param(
    [Parameter (Mandatory=$true)]
    [System.Management.Automation.PSCredential]$Password
)
Write-Host "We'll use this $Password to login."
```

Figure 9.3 Using the System.Management.Automation.The PSCredential method prevents passwords from being exposed.

```powershell
param(
    [Parameter (Mandatory=$true)]
    [System.Management.Automation.PSCredential]$Password
)
Write-Host "We'll use this $Password to login."
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\wcbur\Documents\books\Code\chapter9>
PS C:\Users\wcbur\Documents\books\Code\chapter9> . $args[0] param(
    [Parameter (Mandatory=$true)]
    [System.Management.Automation.PSCredential]$Password
)
Write-Host "We'll use this $Password to login."

cmdlet  at command pipeline position 1
Supply values for the following parameters:
Password
User: bburns
Password for user bburns: ********

We'll use this System.Management.Automation.PSCredential to login.
```

When we run this script without anything supplied, we are prompted for a User. Notice the username is just a string typed in cleartext.

The script then prompts for a Password. This is a PSCredential object and is not displayed or stored in cleartext.

Both of these items are stored in a single PSCredential we store as $Password.

9.2.4 How we use this in our Tiny PowerShell Script

Now that we can securely ask for and pass our credentials we can use this to make our remote PSSession connection to the Exchange Server to gain access to all of the Exchange-related cmdlets.

Figure 9.4 Using the Parameters in the Tiny PowerShell Project code.

Unlike what we did in the previous chapter, however, we are not going to establish an Exchange session for each user we plan on adding to the Domain. By putting this function outside of the loop we can establish the connection once, use it for many users, then disconnect it once.

To see how this works, let's take a quick look at this code with a statement both inside and outside the loop.

```
$list=@(1,2,3,4,5)
Write-Host "This is outside the loop."     #A
foreach ($number in $list){
```

```
    Write-Host "This is within the loop"    #B
}
```

By knowing what information needs to change, and what information is likely not to change during the execution of our script we can design smarter scripts that will prevent us from having to enter the same credentials dozens, hundreds, or even thousands of times when we use this script to create bulk users. If we were to put the Connect-Exchange function within the loop, it would attempt to connect to the exchange server each time the loop was executed. By keeping it outside the loop, we can prevent this behavior by connecting once, then running all of the loops, and finally disconnecting at the end of the script.

9.3 Creating Objects for GUI

As comfortable as I am working within the Command Line Interface (CLI), there are times when having a Graphical User Interface (GUI) object makes things much easier. In our script, we are going to use a comma-separated value (CSV) sheet to help us add a large number of users to our system.

It is fully possible to prompt our user for the path to this value and read in the response with our Read-Host cmdlet. But what if the returned string has a typo?

We can add in some error checking to make sure that the path exists and contains the file. But what if the user types another incorrect path? The idea of having two small errors in file paths that could be on a remote share or several files, or several dozen files deep is not too improbable.

We could loop through our prompt and our error checking, constantly prompting our user for paths. What if the mistake is not a typo but simply confusion, the user typing C:\tmp instead of C:\temp? Or if it were a drive mapping that is disconnected or unmapped? J:\... would not mean anything to our script if “J” were not mapped. This lack of visibility can add to a lot of frustration and wasted time.

What if instead of doing all of this we simply built a Windows Dialog Box that allowed the user to explore their system and select the file with a click?

This capability would instantly clear up some of the above issues or at least give our users a better place to start troubleshooting instead of assuming the issue is a bug within our script.

9.3.1 .Net Assemblies

Remember back in our earlier chapters when we were discussing the power and flexibility of the PowerShell language? PowerShell is powerful and flexible because it is built on the .Net Framework, just like other Microsoft coding languages like C# (C Sharp).

This means that PowerShell can use the .Net Framework just like other programming languages. We are going to use this flexibility to create a Windows Dialogue box and use it to allow our users to select the exact CSV they need.

To start we first need to load our assembly.

```
[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms")
```

On the surface, this line of code doesn't do much. But what did it do? Well, PowerShell by default doesn't load a lot of .Net assemblies. The WinForms assembly is not one of the few loaded by default, so we have to tell PowerShell we want to use it.

What is an Assembly?

An assembly is simply a collection of types and resources designed to work together as a single unit within the .Net framework. It serves as the fundamental unit of deployment within .Net.

9.3.2 Dialog Boxes

With this assembly loaded we can start to use the methods associated with the WinForms assembly to create our dialog box.

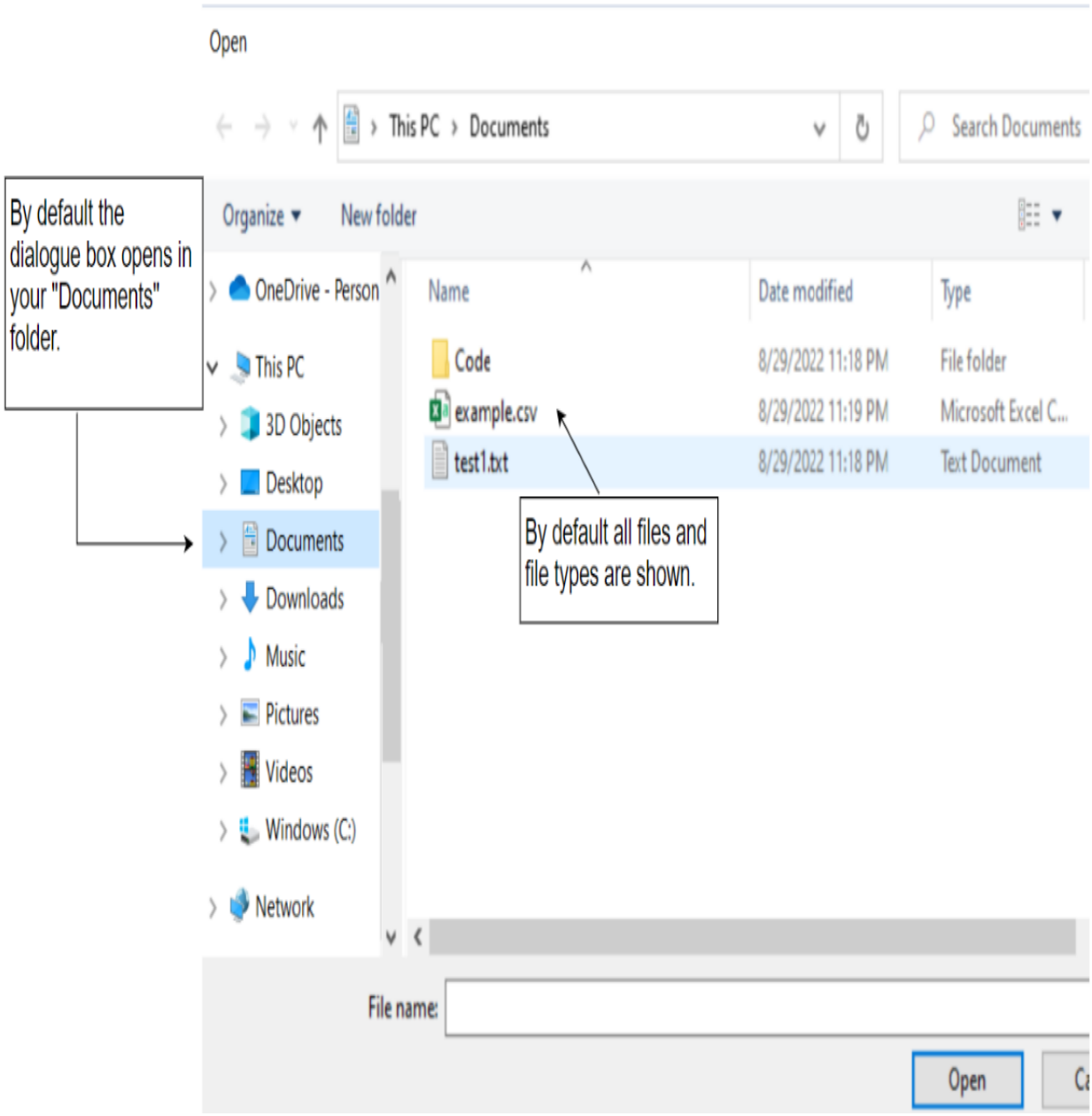
```
[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms")   #A
```

```
$FileDialogBox = New-Object System.Windows.Forms.OpenFileDialog
#B
$FileDialogBox.ShowDialog() | Out-Null#C
```

We simply create a new variable called $FileDialogBox and assign it as a New-Object using the OpenFileDialog method of the WinForm assembly. This saves us a lot of time and effort. We can use the WinForm assembly to create custom forms and objects, with buttons, borders, sizes, and any other attribute of a form.

However, our needs are more simple than this at the moment. We simply want Windows to give us a standard dialog box with an “open” and “cancel” box. This will meet our needs at the moment.

Figure 9.5 Basic dialogue box example.



9.3.3 How we use it in our script

Instead of asking for a user to enter a path to a file we utilize the WinForms Assembly to create a simple dialogue box. However, because our script uses a very specific file type to assist us with user creation, namely the CSV, we want to make sure that the user selects only CSV files.

This can be done with a filter method on the WinForms Assembly dialog box. We also want to make sure that we capture the selected file as a variable we can use.

In our script, we create a function that creates this dialog box, filters for .CSV files, and then returns the filename out of our function into a variable called $CSVLoc

```
Function Get-File(){
[System.Reflection.Assembly]::LoadWithPartialName("System.window
s.forms")
    | Out-Null
    $FileDialogBox = New-Object
System.Windows.Forms.OpenFileDialog
    $FileDialogBox.filter = "csv (*.csv)| *.csv"    #A
    $FileDialogBox.ShowDialog() | Out-Null
         return $FileDialogBox.filename    #B
    }
```

9.4 More Loops

At this point with our PowerShell experience, we are fairly comfortable with loops. We've used them in nearly every script we've made since chapter 4. But there are many different kinds of loops. The one we're most familiar with is the foreach loop. We can use a foreach loop and an if statement to make some type of decision based on the value of the item within the loop.

Say, for example, we wanted to get every bite-size up to 128:

```
$bits=(8,16,32,64,128,256,512)
foreach ($size in $bits){
    if ($size -le 128{
        Write-Output $size
    }
}
```

This kind of logic is an excellent discriminator if we don't know what the sequence of the values in our array are. In this case, however, we know that the numbers double in size each time. So, instead of pulling in each value and testing it against the conditional logic of an if statement, what if we, instead, built the logic into the loop itself?

This is where a While loop comes in.

9.4.1 While Loop

The while loop executes the commands within the loop while the conditional logic within the while statement evaluates as true. It can be thought of as "While something is true, do something." Once the while statement's conditional logic no longer evaluates as true the loop no longer runs.

```
$byte=8    #A
while ($byte -lt 256) {    #B
    Write-Output $byte   #C
    $byte=$byte*2
}
```

This type of logic can be useful for math, but it can be even more useful to check to see if a condition exists. Let's look at some input validation.

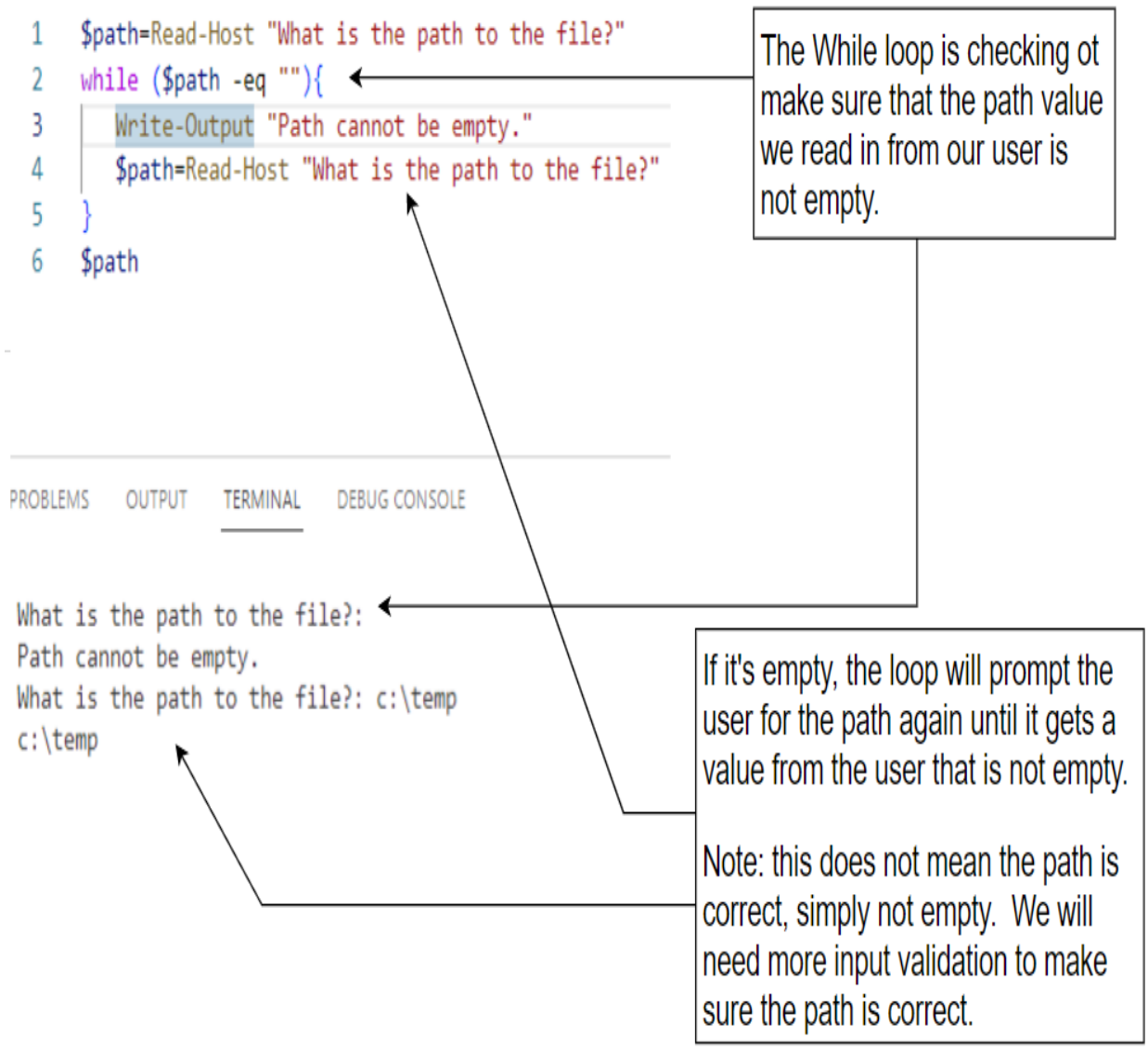
Definition

Input validation is simply the **process** that we employ to test the input received by our application for compliance and compatibility. Input validation can help make sure that the type of data we are expecting is what we receive. In many cases, the type of data we receive can be important. We've already discussed data types like strings and integers in previous chapters. Input validation can help us make sure we get the type of data we expect and can prevent significant errors later in our code.

We can use the While loop to make sure that the user enters a path for us to use later in our script.

```
$path=Read-Host "What is the path to the file?"
while ($path -eq ""){
   Write-Output "Path cannot be empty."
   $path=Read-Host "What is the path to the file?"
}
$path
```

Figure 9.6 An example of input validation for a non-empty string.



9.4.2 How we use this in our script.

Because we require the CSV file to add our users to Active Directory we need to make sure that our script is pointing to the CSV our user intends. We have already seen how we use the Windows Dialog box to aid our users in navigating directly to the file. However, the dialog box included both an "Open" button and a "Cancel". If the User were to click cancel we'd want a way to detect that and re-prompt our user for the correct file.

We use the While loop to re-prompt our user for the file location whenever a null or empty value is returned for the location of the CSV file.

```
while ([string]::IsNullOrEmpty($CSVLoc)) {    #A
    Write-Output "Navigate to the CSV file to create your
users:"
    $CSVLoc=Get-File    #B
}
```

9.5 IsNullOrEmpty

As you may have noticed in the while loop above we are using more .Net. This time, we are using a string method IsNullOrEmpty. This method will test to see if the string value of the object is either null or empty.

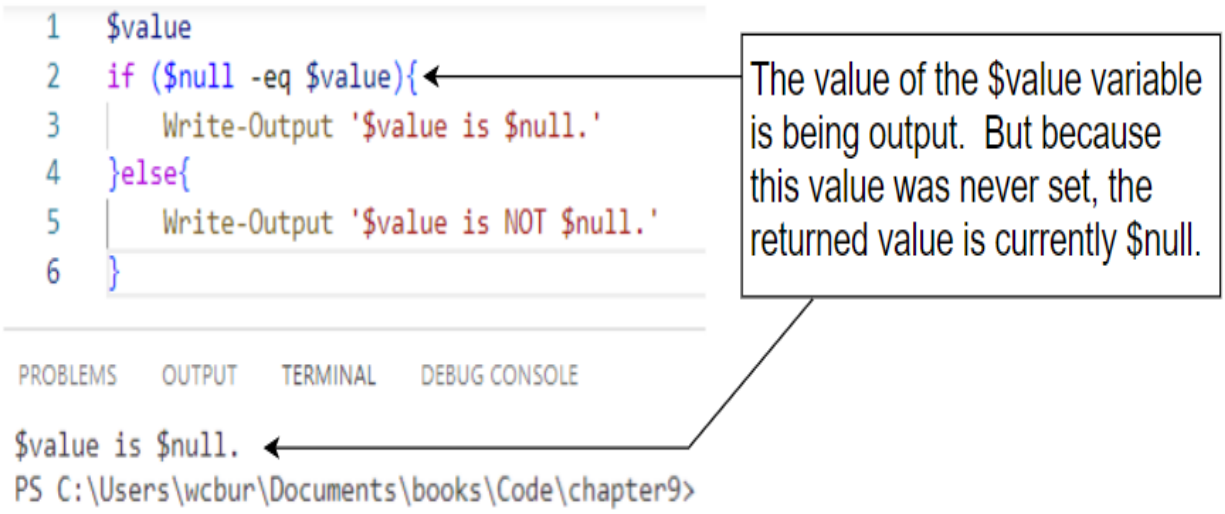
9.5.1 Differences between Null and Empty

When we were discussing null values in previous chapters we touched on what a null value was. But it can be very confusing to distinguish between a null value and an empty value.

A null value is a value that is missing or unknown. It can be the value of a variable that is either intentionally set to $null or simply not holding any value.

```
$value
if ($null -eq $value){
    Write-Output '$value is $null.'
}else{
    Write-Output '$value is NOT $null.'
}
```

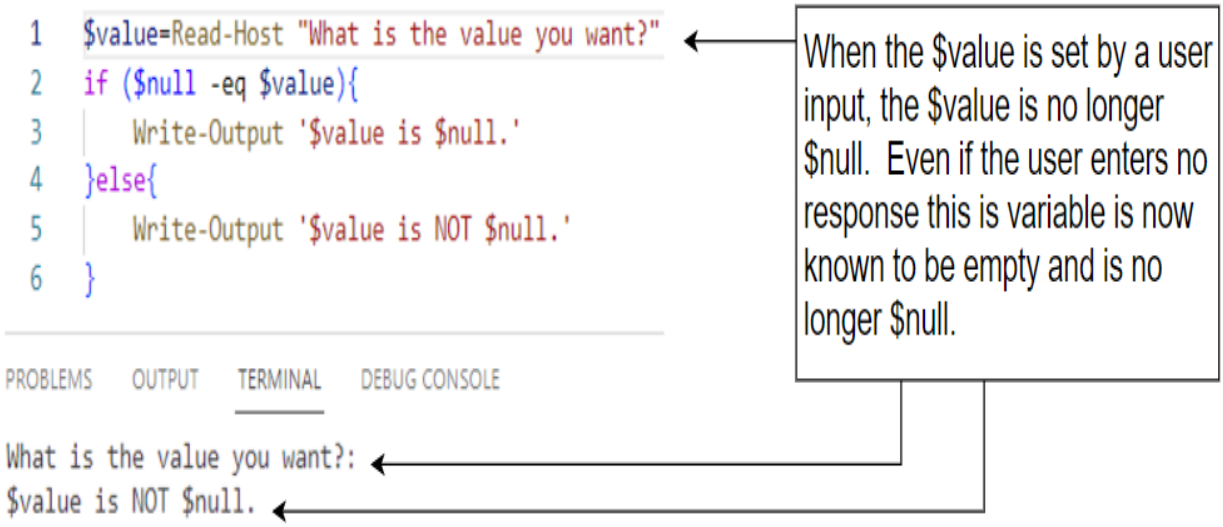
Figure 9.7 A null value example.



However, an empty value is not null, it is empty instead. If we prompt our user for the $value, any input, even a blank value, will technically no longer be null.

```
$value=Read-Host "What is the value you want?"
if ($null -eq $value){
    Write-Output '$value is $null.'
}else{
    Write-Output '$value is NOT $null.'
}
```

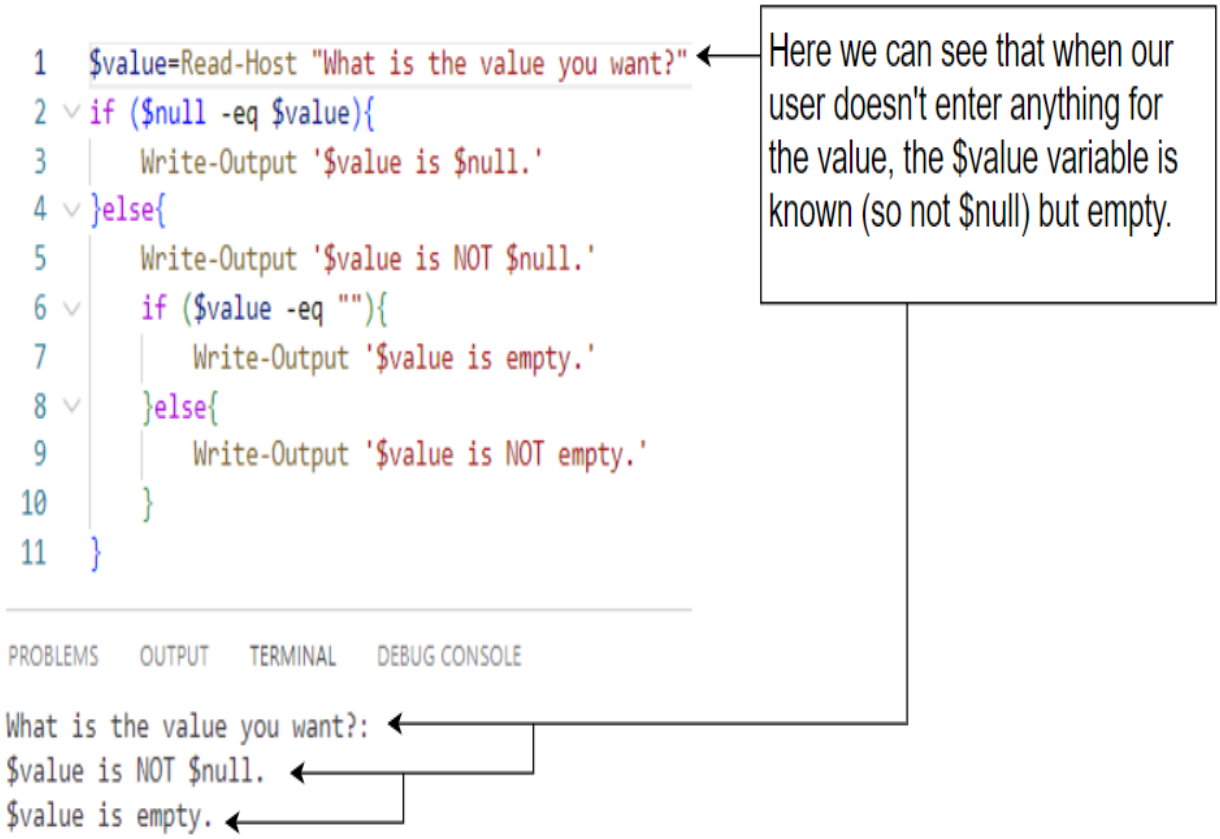
Figure 9.8 A non-null example.



We can check for empty values with another if statement in our code:

```
$value=Read-Host "What is the value you want?"
if ($null -eq $value){
    Write-Output '$value is $null.'
}else{
    Write-Output '$value is NOT $null.'
    if ($value -eq ""){
        Write-Output '$value is empty.'
    }else{
        Write-Output '$value is NOT empty.'
    }
}
```

Figure 9.9 Example of a non-null, but empty string.



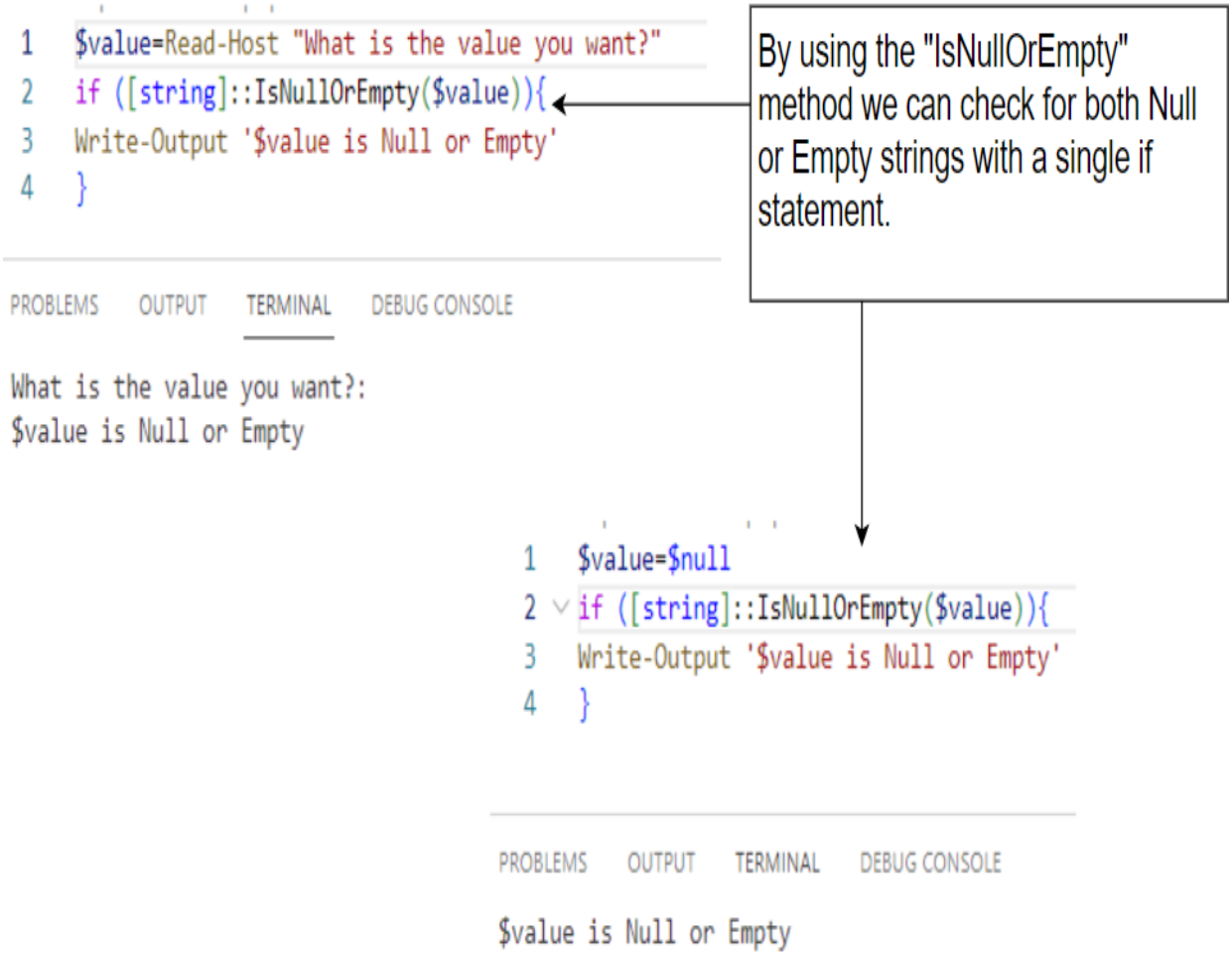
9.5.2 Checking for both nulls and empties

By using the "IsNullOrEmpty" method for our string value we can use a single check to check for either condition. Not only is this less code to type,

but it's also easier to understand. The nested logic of the if/else statements can be confusing and difficult to debug.

```
$value=$null
if ([string]::IsNullOrEmpty($value)){
    Write-Output '$value is Null or Empty'
}
$value=Read-Host "What is the value you want?"
if ([string]::IsNullOrEmpty($value)){
    Write-Output '$value is Null or Empty'
}
```

Figure 9.10 IsNullOrEmpty finds both null and empty strings.



9.5.3 How we use it in our script.

It's likely that when the script is run initially, the CSVLoc variable will not be defined. This will produce a $null value for the variable. Because this path is needed for the rest of the script to work correctly, we want to trigger the Get-File function to prompt our user to explore to the CSV file location.

But as a part of our input validation, what happens if our user closes the window or hits cancel before they select the CSV? In this case, the value would be null but instead empty.

Instead of writing nested if/else statements we can put a single check within our While loop to continue to prompt our user for a valid CSV location until they successfully explore to the file and click on "open".

```
while ([string]::IsNullOrEmpty($CSVLoc)) {    #A
    Write-Output "Navigate to the CSV file to create your
users:"
    $CSVLoc=Get-File
}
```

9.6 Import-CSV

In its simplest form, a CSV is a file that contains values separated by commas. By using the Import-CSV cmdlet we instruct PowerShell to create a collection of custom objects from a CSV file. Because of the file type, programs that open the CSV file type recognize the structured data instead of simply displaying the raw (comma-separated format). This allows for tables to be easily constructed and manipulated.

Raw CSV files can be read in any text editor, but the real power and flexibility of the file type is when you use spreadsheet programs like Excel to create or modify them.

```
"President, Order, Years
George Washington, 1, 8
John Adams, 2, 4
Thomas Jefferson, 3, 8
James Madison, 4, 8"|out-file c:\temp\presidents.csv
```

By writing the output and separating the values by commas and the rows by lines we're able to use PowerShell to create a CSV.

Figure 9.11 The CSV file we created can easily be opened in Excel.

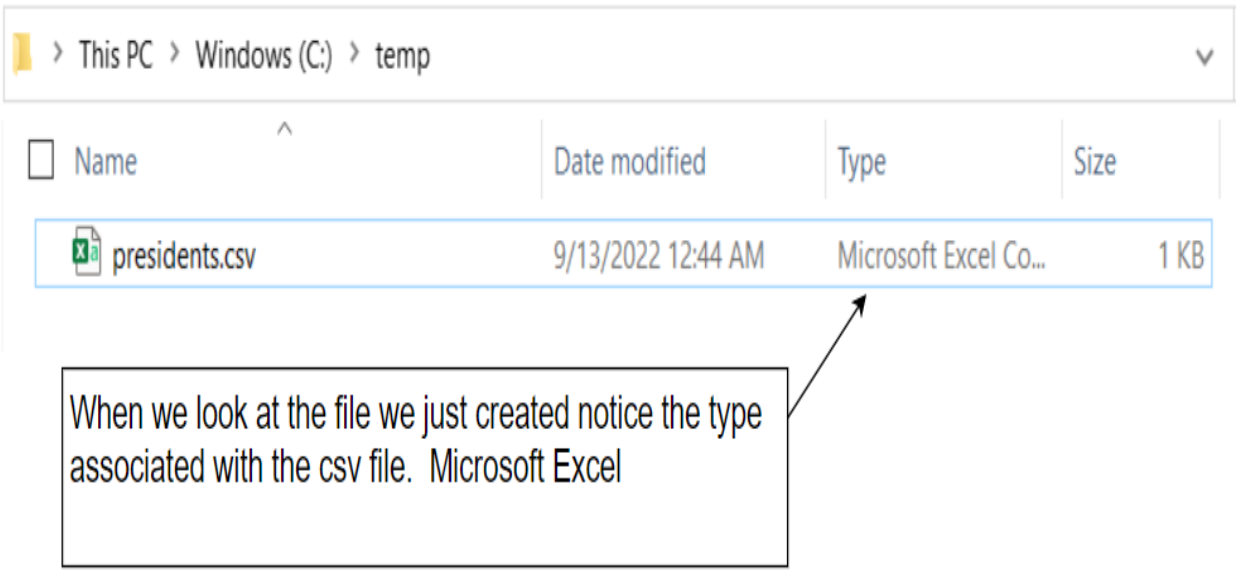
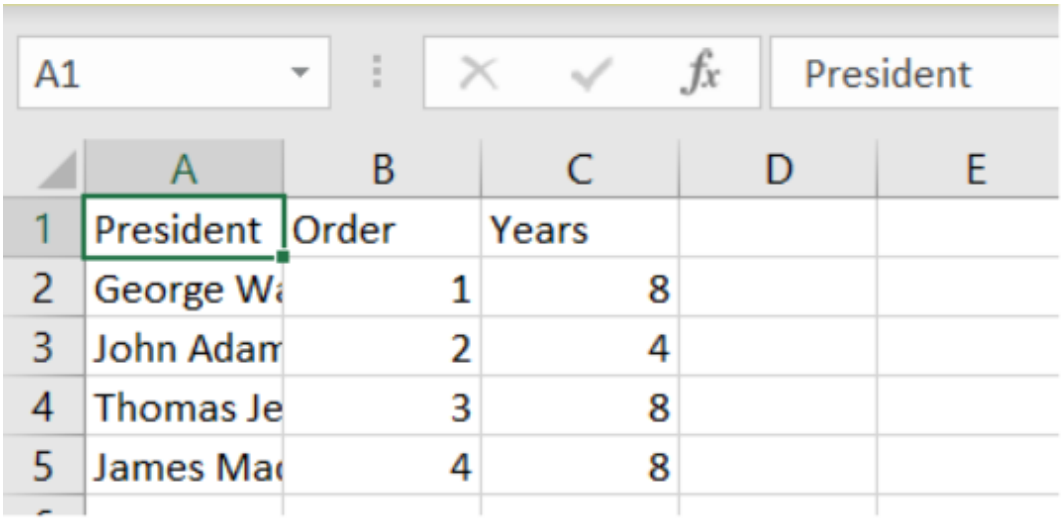


Figure 9.12 Opening the CSV in Excel no longer contains any commas within the data.



	A	B	C	D	E
1	President	Order	Years		
2	George W	1	8		
3	John Adam	2	4		
4	Thomas Je	3	8		
5	James Mac	4	8		

The file we created is opened and understood by Excel. All of the formulas, macros and imports that you are capable of doing in Excel are now at your fingertips.

You can modify the CSV in Excel by simply entering data in the coresponding cell. Excel will insert the comas in the backend.

You can also save .xlsx files as .csv files with a "save as" function in excel.

Excel can be used to manipulate the data, perform calculations, separate text, merge cells and all of the many powerful functions people use Excel for every day. I was able to update my meager president list by copying and pasting the data found at the Library of Congress. I then broke down the starting and ending years into separate columns and used a calculation function to generate the term. Then resaved the data as presidents2.csv.

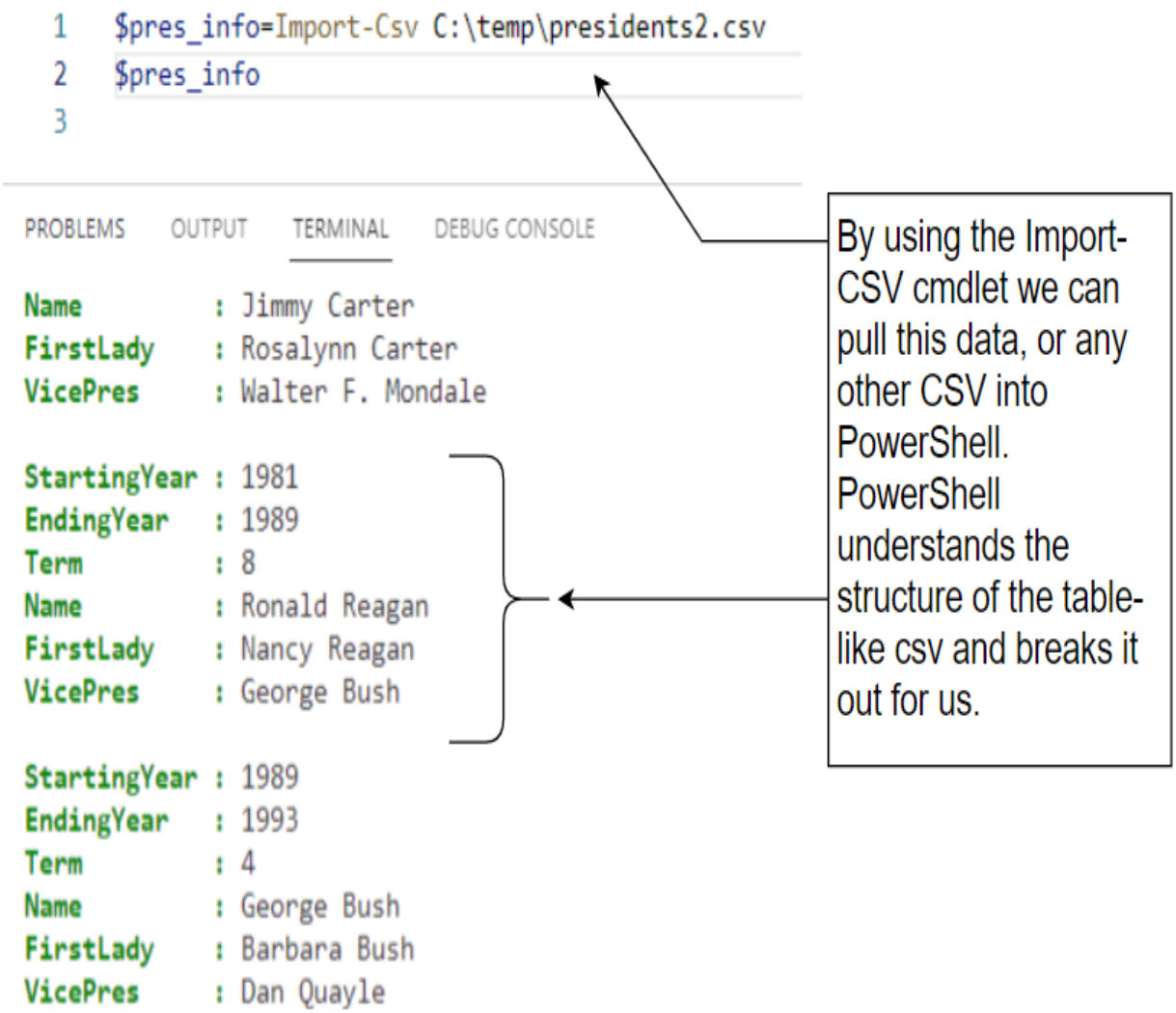
Figure 9.13 is an example of the power of a CSV.

StartingYear	EndingYear	Term	Name	FirstLady	VicePres
1789	1797	8	George Washington	Martha Washington	John Adams
1797	1801	4	John Adams	Abigail Adams	Thomas Jefferson
1801	1805	4	Thomas Jefferson	[Martha Wayles Skelton Jefferson	Aaron Burr
				died before Jefferson assumed office;	
				no image of her in P&P collections]	
1805	1809	4	Thomas Jefferson	see above	George Clinton
1809	1812	3	James Madison	Dolley Madison	George Clinton
1812	1813	1	James Madison	Dolley Madison	office vacant

With the power of the internet and the power of Excel I was able to quickly and easilly find information for every president to date. I was able to also seperate the starting year and the ending year and calculate the term in office. This would have been a lot of typing by hand!

Even with all of the links, merging, calculations, and other Excel manipulations, the CSV is still just a file of comma-separated values. These values can be imported into PowerShell with the Import-CSV cmdlet and because it is a CSV, PowerShell natively reads the data as structured data.

Figure 9.14 Importing the CSV brings the data structure into PowerShell.



9.6.1 How we use it in our script

To generate a new user we need to supply our script with three basic pieces of information: A first name, a last name, and a user we want to copy permissions from. Because of the ubiquities of Excel in business today, likely, the first and last names of the users we're being asked to create are already in an Excel format.

We can easily create a CSV in Excel that includes the FirstName, LastName, and CopyFromUser information for each of the users we are going to create.

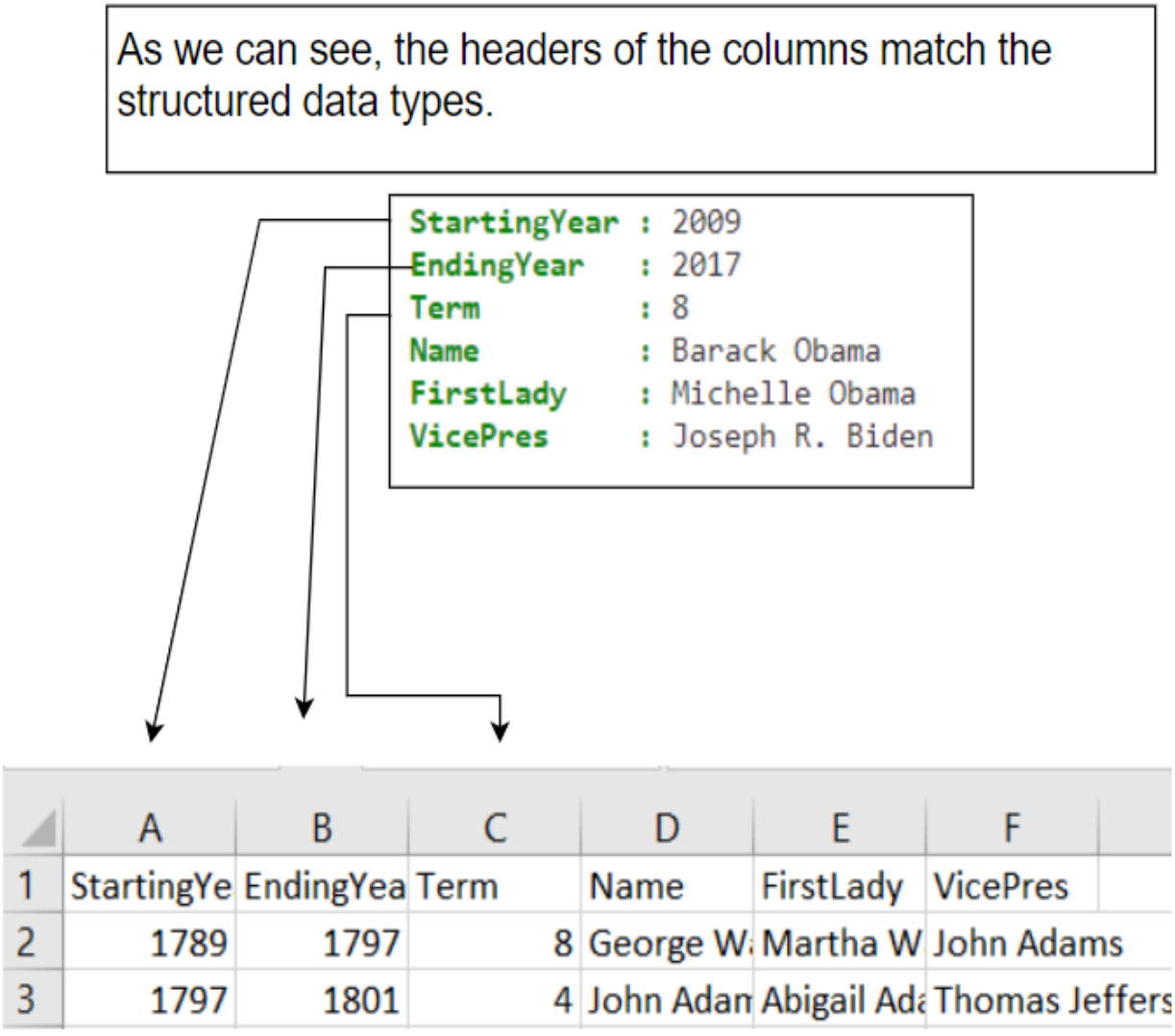
The Import-CSV cmdlet will be used to help PowerShell understand the structure of this data so we can use it to create the users.

```
$CSVInfo=Import-Csv -Path $CSVLoc
```

9.7 Targeting within the CSV

One of the benefits of PowerShell understanding this structured data is we can use the header information to target specific cells.

Figure 9.15 PowerShell breaks up the data by headers.

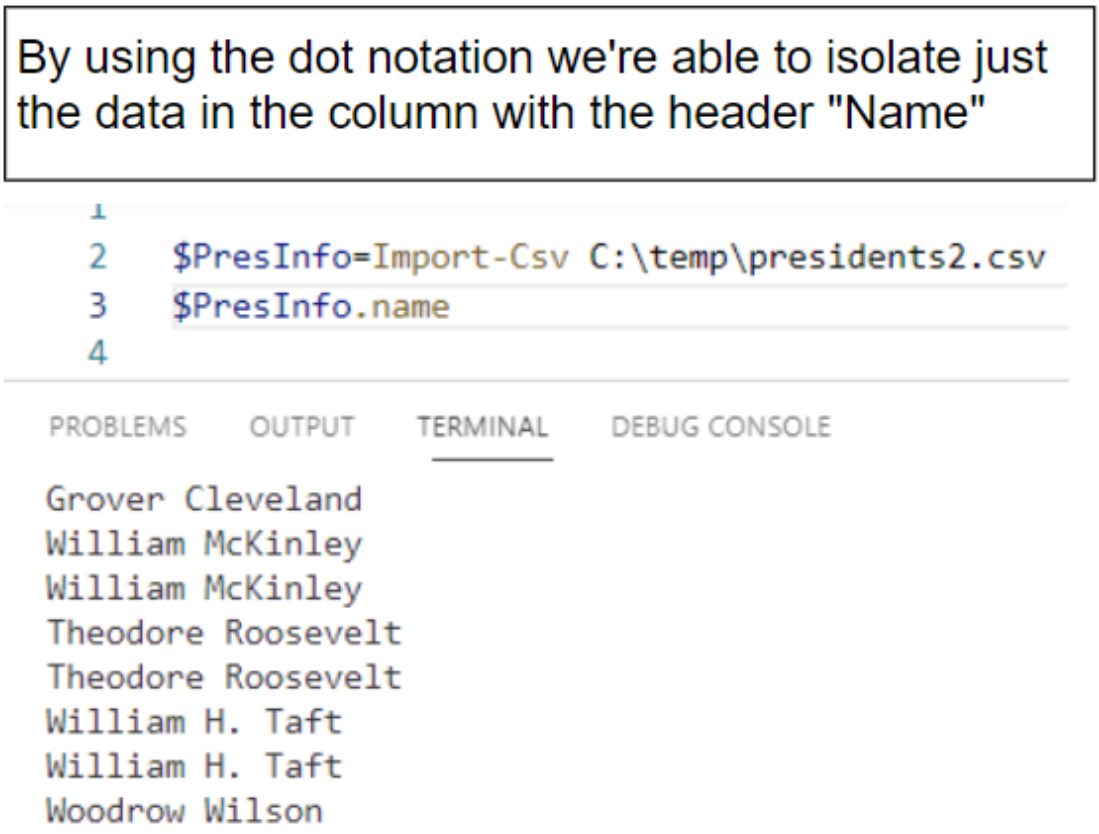


PowerShell reads the data and creates properties for each of them. The “Term”, “Name”, “StartingYear”,… etc. are each treated as properties for each object within our collection.

We can use the dot notation to reference just the values in a single column.

```
$PresInfo=Import-Csv C:\temp\presidents2.csv
$PresInfo.name
```

Figure 9.16 Using the dot notation to select a column.



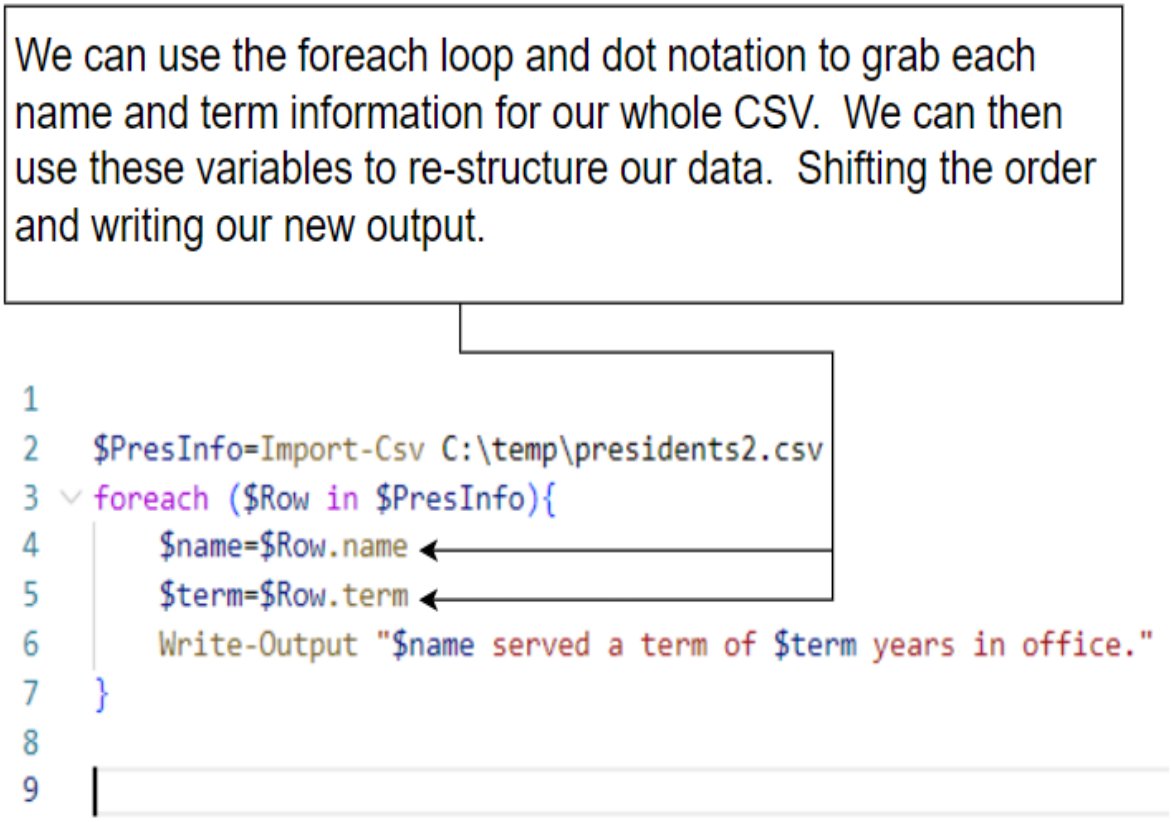
When we assign a collection of objects to our variable “PresInfo”, we can then use the dot notation to isolate specific properties for each object within that collection of objects.

By looping through the data we can isolate each cell and manipulate the contents of any given cell. Say, for example, we only cared about the President’s name and term of office. We could modify the CSV file

eliminating these columns. But there may be a time we need this data. Or we may not have permission to modify the original file. With PowerShell, we can isolate and display the values we want and order them in any way we wish.

```
$PresInfo=Import-Csv C:\temp\presidents2.csv
foreach ($Row in $PresInfo){
    $name=$Row.name
    $term=$Row.term
    Write-Output "$name served a term of $term years in office."
}
```

9.17 using foreach and dot notation to target specific cells.



9.7.1 How we use this in our script

To create our user, we need three pieces of information:

- A First Name
- A Last Name

- A User to Copy Group Membership From

By collecting this information for each new user we want to create and storing it in a CSV file we can use this file as the input source of this information for our script.

This allows us to simply point PowerShell to the location of the information and PowerShell will import the CSV and use the dot notation to pull the pieces of data that it needs to create the users for each entry in our CSV. Do keep in mind these headings need to correspond exactly with the dot notation you use to reference them.

```
$CSVInfo=Import-Csv -Path $CSVLoc    #A

foreach ($user in $CSVInfo){
    $LastName=$user.LastName    #B
    $FirstName=$user.FirstName    #B
    $CopyFromUserSamAccountName=$user.CopyFromUser
```

9.8 Summary

In our previous chapter we developed a script that allowed us to create Windows Domain accounts with ease. Using the script required that the operator provide the script a First Name, Last Name, and User to Copy group membership from. This is great when creating a single user! But running the script over and over and manually providing the First Name, Last Name, and User to Copy from was tedious. Instead of entering this information one at a time, we can create a CSV file with this information for every user we want to create leveraging Excel's many powerful spreadsheet tools. With the use of a CSV file that includes a First Name, Last Name, and User to copy permissions from, we can create dozens, hundreds, or even thousands of users on our domain with a single script.

- Using System.Management.Automation.PSCredentials allow us to collect usernames and passwords in a secure and convenient storage object called a PSCredential object.
- With .Net Assemblies we can construct custom objects from scratch giving PowerShell the flexibility of the whole framework similar to C#.

- Using one of these .Net Assemblies (WinForms) we can make Windows Dialog boxes to allow our user to explore to input files with a GUI
- The While loop executes the instructions within the loop while a condition is $True. After each execution of the loop, it tests the condition, and if the condition no longer evaluates as $True, the loop exits.
- There is a difference between a null value and an empty value. Checking for both of these typically requires two separate checks. We learned how to use .Net to check for an IsNullOrEmpty condition allowing for a single check for both conditions.
- Using the Import-CSV cmdlet we can import a collection of objects with properties drawn from our CSV structure. This allows us to easily target specific properties within our collection for use in our script.
- With the dot notation we can target specific properties of our object collection imported from a CSV file. This allows us to create a single CSV to create dozens, hundreds, or even thousands of users with a single execution of our script.

10 Inventory expert

This chapter covers:

- Querying Active Directory for enabled computers
- Quickly checking whether computers are online
- Checking a variable for multiple different values without nested if/else statements
- Gathering information from remote computers
- Outputting arrays to CSVs

If you've been in the IT industry long, you realize there is a lot of data to manage. Each computer has dozens of components and characteristics that are valuable. If a new virus, for example, affects only specific versions of Windows 10 it can be a mad dash to figure out exactly how many computers in your company are at risk and need to be mitigated and accounted for.

Hard Drive space can fill up rapidly, and critical data may be irrevocably lost once a drive is full. As PCs age, you may need to upgrade RAM to keep the system functional. Serial numbers are often used by vendors for support, and for remote computers, in the past, gathering all of the required data meant a phone call at best and perhaps even a trip to the location at worst to get this information and more. Worse, if you didn't gather all of the data or recorded it incorrectly this could mean a costly re-work.

Wouldn't it be nice if we could simply run a script and gather all of this information and more from any computer, a list of computers (say all of the PCs in a particular geographic location), or all of the active computers on your domain? This is exactly the need we're going to address in this very chapter!

10.1 Project Code

The data-gathering code is below; like our other scripts, we'll cover it in detail. There are a few special areas to take note of, detailed below.

```
Param(
    [Parameter(Mandatory=$true)][string]$SavePath,
    [Parameter(Mandatory=$false)][string]$LookUpType
)
Function Get-File(){

[System.Reflection.Assembly]::LoadWithPartialName("System.window
s.forms")
        | Out-Null
        $FileDialogBox = New-Object
System.Windows.Forms.OpenFileDialog
        $FileDialogBox.filter = "txt (*.txt)| *.txt"
        $FileDialogBox.ShowDialog() | Out-Null
             return $FileDialogBox.filename
        }

while ([string]::IsNullOrEmpty($LookUpType)) {
    Write-Host "Do you want to query a (s)ingle PC, a (l)ist of
PCs,`
                or (a)ll PCs on your domain:"
    $LookUpType=Read-Host "s, l, a: or any other key to cancel?"
}

switch ($LookUpType[0]) {    #A
    "a"     {$ObjList=(Get-ADComputer -filter *|Where-Object `
             {$_.enabled -eq $true}).name  }
    "s"     {$ObjList=Read-Host "Enter the name of the PC"  }
    "l"     {$TargetList=Get-File;
             $ObjList=Get-Content $TargetList  }
    Default {write-host "No valid LookUpType found, please try
again and`
             select : a, s, or l"; exit}
}

foreach($PC in $ObjList){
    $Alive=Test-Connection -Count 1 $PC -Quiet    #B
    if ($Alive){
        $Online="True"
        $base= Get-CimInstance -computername $PC -ClassName `
#C
               Win32_ComputerSystem -ErrorAction
SilentlyContinue
        $os = Get-CIMInstance -computername $PC -class `
              Win32_OperatingSystem -ErrorAction
SilentlyContinue
        $vol = Get-CIMInstance -computername $PC -class
Win32_Volume|
               Where-Object {$_.DriveLetter -eq "C:"}`
```

```
                -ErrorAction SilentlyContinue
        $net = Get-CIMInstance -computername $PC -class `
                Win32_NetworkAdapterConfiguration |
                where-object { $_.IPAddress -ne $null } -
ErrorAction `
                SilentlyContinue

    }else{
        $Online="False"
    }


        $DeviceInfo= @{    #D
            Online=$Online
            SystemName=$PC
            OperatingSystem=$os.name.split("|")[0]
            SerialNumber= $os.SerialNumber
            Version=$os.Version
            Architecture=$os.OSArchitecture
            Organization=$os.Organization
            Domain=$base.Domain
            RAM_GB=$base.TotalPhysicalMemory/1GB
            IPAddress=($net.IPAddress -join (", "))
            Subnet=($net.IPSubnet -join (", "))
            MACAddress=$net.MACAddress
            DiskCapacity_GB= $vol.Capacity/1GB
            FreeCapacity_GB=$vol.FreeSpace/1GB
        }
while ($SavePath -notlike "*.csv"){
    Write-Error "The save path must end in a csv file name `
                `n Example: c:\temp\report.csv"
    $SavePath=Read-Host "Enter the full path and name of the csv
to save`
                        the data to"
}

$DeviceInfo| Export-CSV $SavePath -Append    #E
    }
```

10.2 The Switch Statement

We have been using conditional logic within our scripts for some time. If, Elseif, and Else statements allow us to make decisions within our scripts. The structure of these statements makes them ideal for choices of three or fewer.

```
$Fruit=Read-Host "Pick a fruit"
if ($Fruit -eq "banana"){
    write-host "a $fruit is yellow"
}elseif ($Fruit -eq "apple") {
    write-host "an $fruit is red"
}else{
    write-host "I don't know what color a/an $fruit is."
}
```

However, when you introduce more than three choices it gets complex and ugly quickly.

```
$Fruit=Read-Host "Pick a fruit"
if ($Fruit -eq "banana"){
    write-host "a $fruit is yellow"
}elseif ($Fruit -eq "apple") {
    write-host "an $fruit is red"
}elseif ($Fruit -eq "orange") {
    write-host "an $fruit is orange"
}elseif ($Fruit -eq "grape") {
    write-host "a $fruit is green"
}elseif ($Fruit -eq "cherry") {
    write-host "a $fruit is red"
}elseif ($Fruit -eq “kiwi”) {
    write-host “a $fruit is green”
}else{
    write-host “I don’t know what color a/an $fruit is.”
}
```

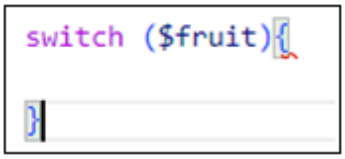
The Switch statement is a more concise way to check for multiple conditions without nested if/else statements.

```
$Fruit=Read-Host “Pick a fruit”
switch ($Fruit) {
“banana” {write-host “a $fruit is yellow”}
“apple”  {write-host “an $fruit is red”}
“orange” {write-host “an $fruit is orange”}
“grape”  {write-host “a $fruit is green”}
“cherry” {write-host “a $fruit is red”}
“kiwi”   {write-host “a $fruit is green”}
    Default {write-host “I don’t know what color a/an $fruit
is.”}
}
```

10.2.1 Structure of the Switch Statement

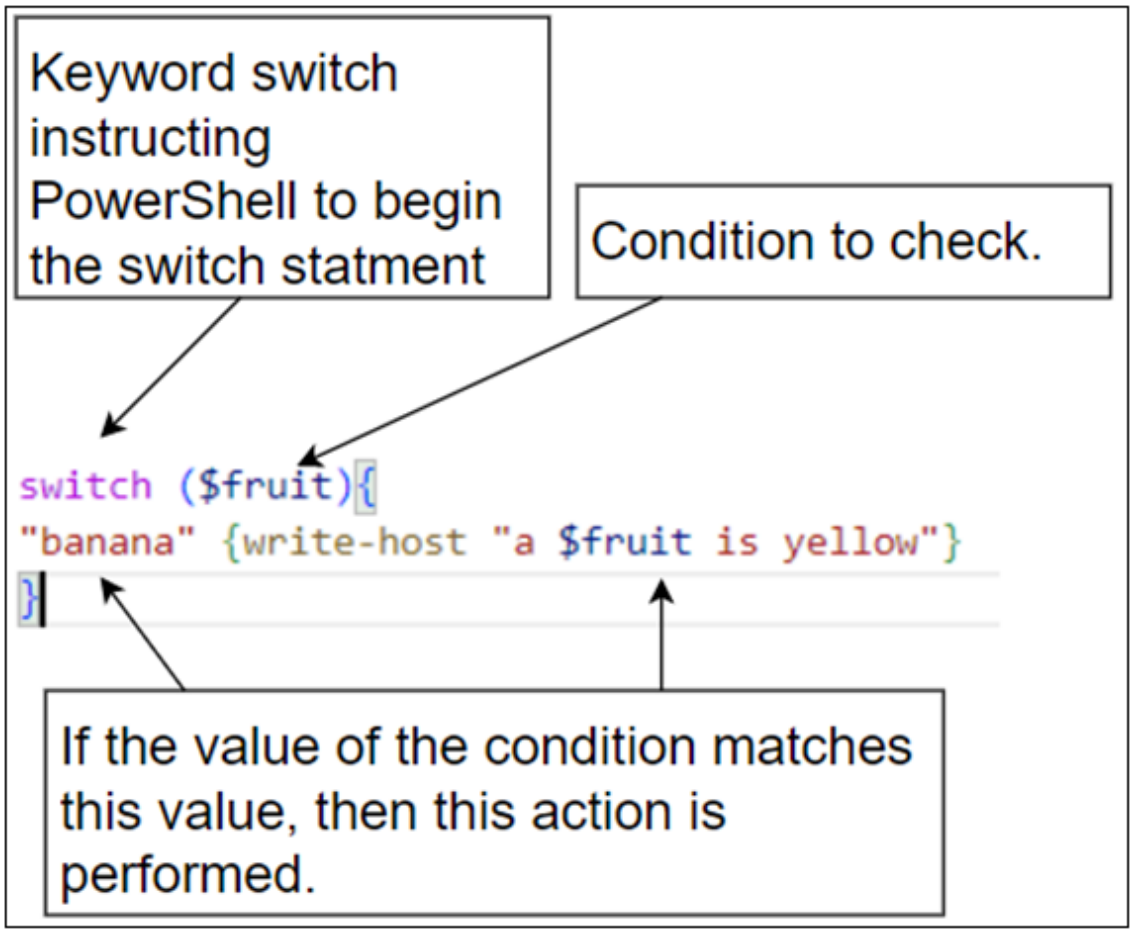
In PowerShell, a Switch is defined with the keyword `switch` followed by the expression to check within parentheses. The switch actions are then encapsulated within curly braces.

Figure 10.1 Switch framework



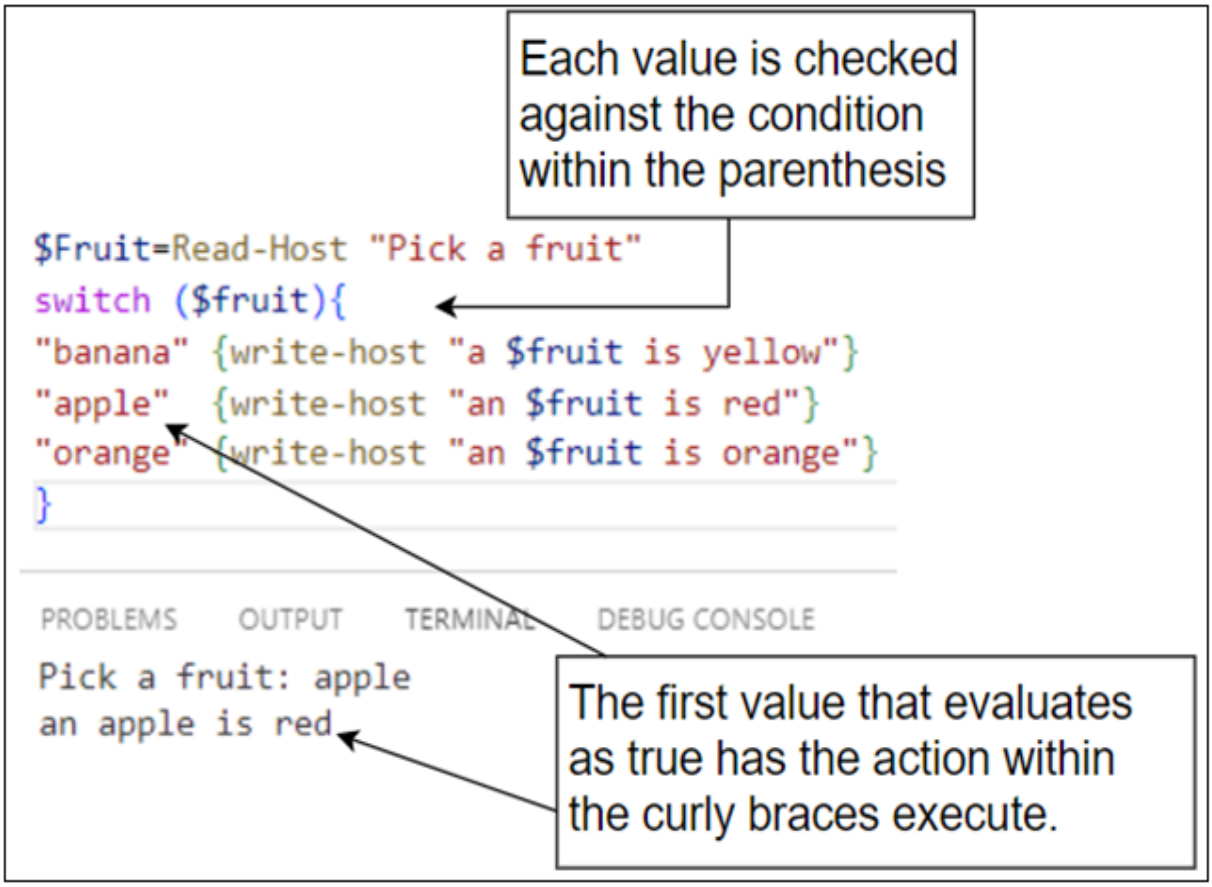
Within these curly braces, the possible matches for the condition are then defined in quotation marks, each followed by the action that should be taken if that match equals the condition's value, each action is defined by its own set of curly braces.

Figure 10.2 Checking values within a switch statement.



Multiple values for the condition can be defined and checked within the switch statement. Each subsequent value checked is listed within the curly braces of the switch statement. The first value that evaluates as true is then executed.

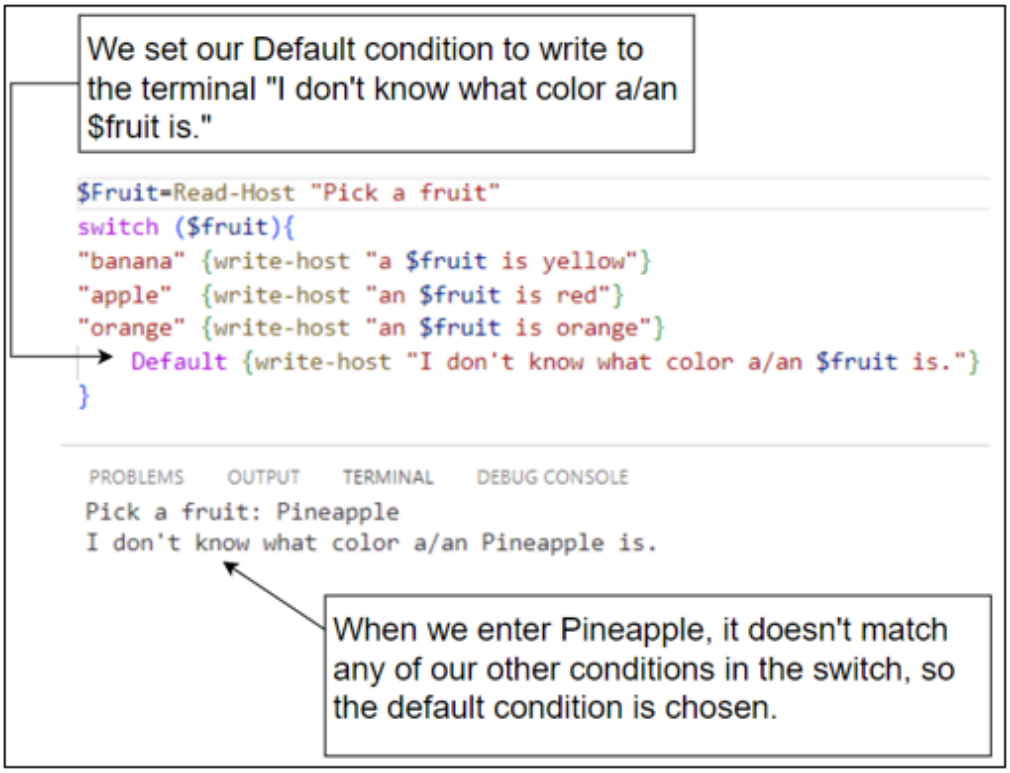
Figure 10.3 Checking condition against multiple values in a switch statement.



Note

When using switch statements it's best practice, to establish a default action. A default action will happen when none of the other conditions resolve to true.

Figure 10.4 Using a default condition in a switch.



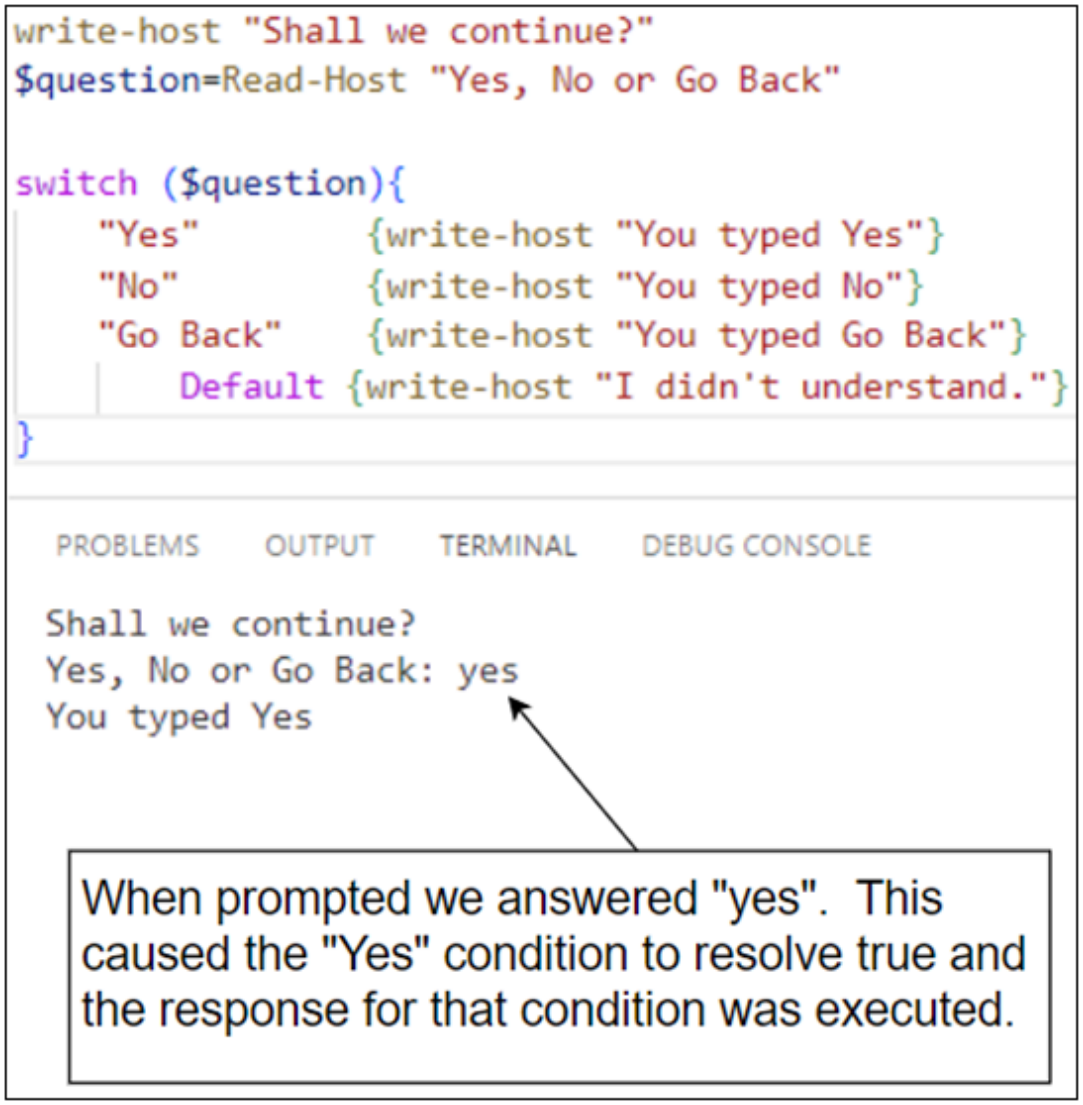
10.2.2 Selecting the first character for switches

Users can be unpredictable; thus, we may be unsure of exactly what input they are going to type. In our Tiny PowerShell project, we are building a menu and asking our user to navigate it.

```
write-host "Shall we continue?"
$question=Read-Host "Yes, No or Go Back"

switch ($question){
    "Yes"         {write-host "You typed Yes"}
    "No"          {write-host "You typed No"}
    "Go Back"     {write-host "You typed Go Back"}
        Default {write-host "I didn't understand."}
}
```

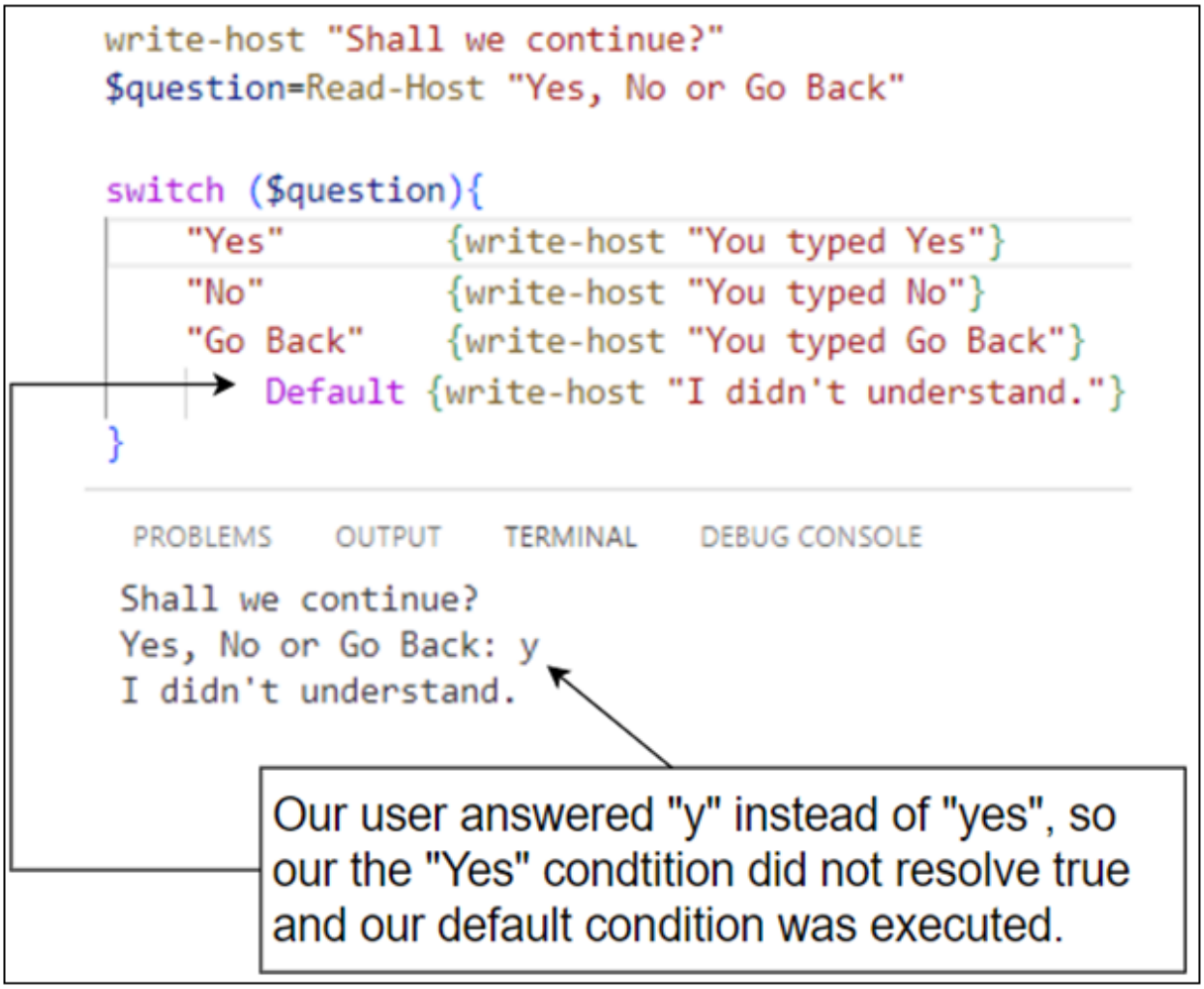
This is a simple example of a menu. It prompts our user for a Yes, No, or Go Back response. If it doesn't get the answer it expects, it tells the user that it didn't Understand.

Figure 10.5 Switch Statement gets the response it expects.



But what happens if the user simply types “Y”?

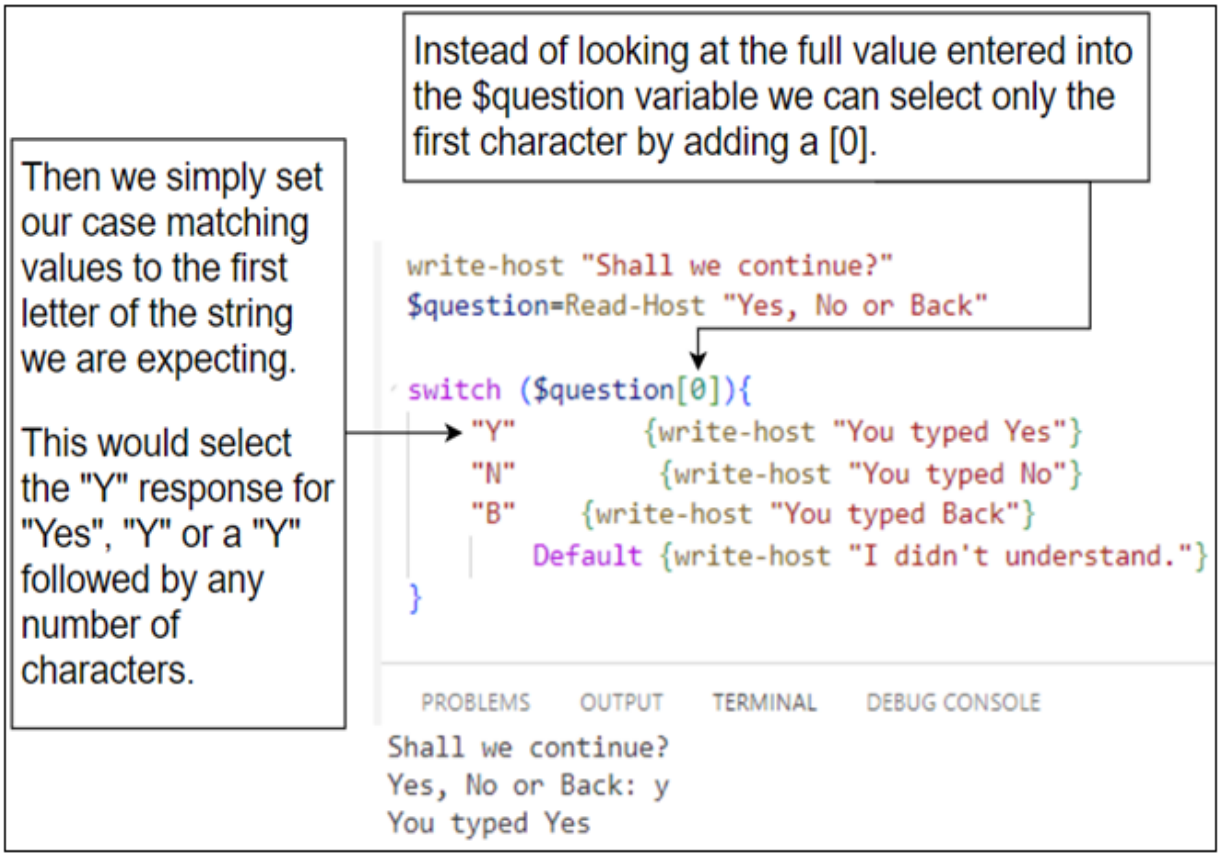
Figure 10.6 Switch Statement not getting the response it expects.



Because our condition does not use wildcards the switch looks for exact matches (non-case sensitive) for the condition. “yes” or “Yes” would work, but as we’ve seen “y” does not. This type of behavior for our menus could seem frustrating for our users since they are constantly typing out full words. We can fix this by simply checking the first character of the response. This will evaluate “yes” and “y” the same.

```
write-host "Shall we continue?"
$question=Read-Host "Yes, No or Back"

switch ($question[0]){
    "Y"          {write-host "You typed Yes"}
    "N"          {write-host "You typed No"}
    "B"          {write-host "You typed Back"}
    Default      {write-host "I didn't understand."}
}
```

Figure 10.7 Example of the switch statement with only the first character.



10.2.3 How do we use this in our Tiny PowerShell Script?

First, we create a variable called $LookUpType. We include this as one of the parameters for the script so a user can include the desired number of computers to be included in our inventory (one, several, or all) as soon as they initialize the script.

If, however, this is not passed as a parameter when initializing the script, we set up a While Loop, and as long as the $LookUpType variable is null or empty it will prompt our user to supply this information.

Finally, we check the value of the $LookUpType with a switch statement. We select the first character of the response using the [0] array notation in case our user typed “all” instead of “a”, for example. The value returned in

the $LookUpType variable will determine how many, and which, PCs we are pulling the hardware information from.

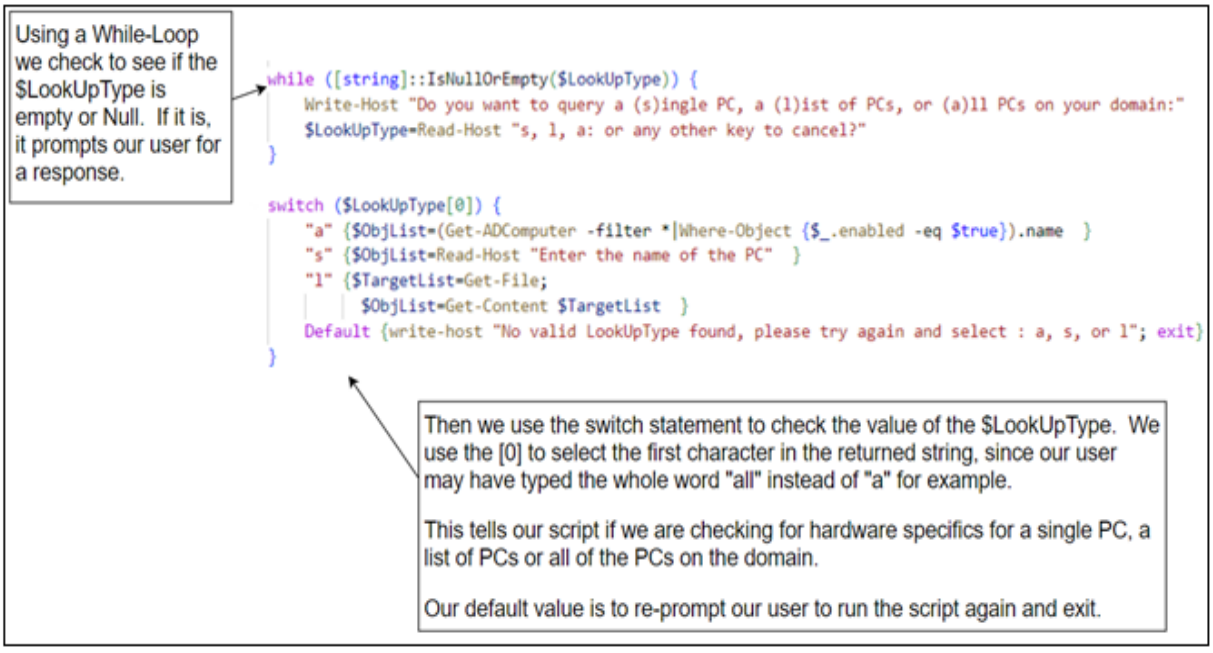
If the user enters “a” or any string that starts with an “a” the script will perform an Active Directory query for the names of all enabled computers on the domain and set the $ObjList variable equal to this list.

If the user selects “s” or any string that starts with an “s” the script will prompt the user for the name of the PC they wish to check and set the $ObjList variable equal to this value.

If the user selects “l” or any string that starts with an “l” the script will create a variable $TargetList and then run the Get-Content function. This function will open up a GUI menu that will allow the user to select the .txt file containing the list of PCs. The script will then read the contents of this file and set the $ObjList variable equal to this content.

If the user presses any other key or combination of keys, the script will prompt the user that no valid LookUpType was found and exit the script.

Figure 10.8 Using the switch statement in our Tiny PowerShell project.



10.3 Test-Connection

When you are dealing with objects at a large scale, performance becomes an issue we must pay special attention to. If a script takes a few seconds to run on each system or object, it can add minutes, hours, or even days to a large enough collection. One thing that commonly adds several seconds is a remote computer that is not reachable remotely. PowerShell will attempt to reach the system and wait until the command times out before moving on. Below we can see the impact this has on the runtime of our scripts

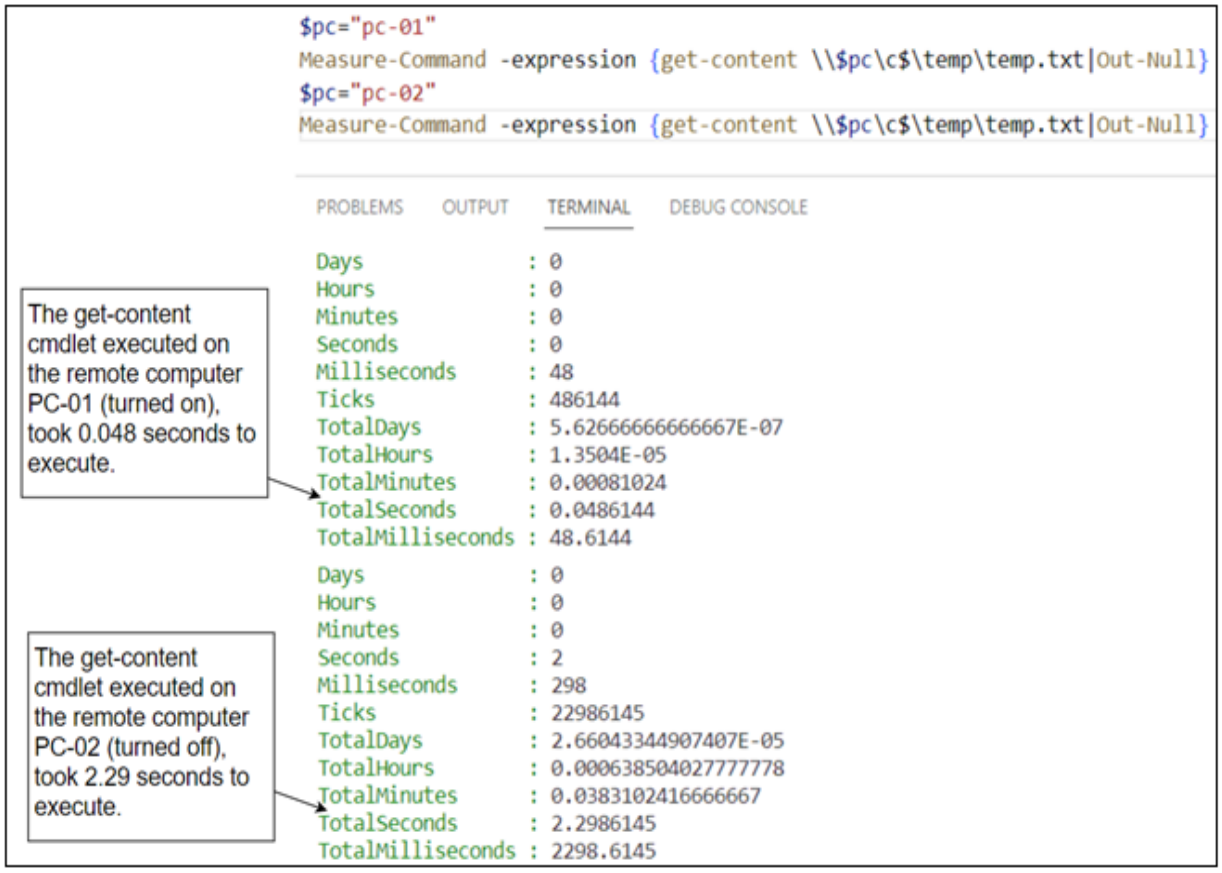
Measure-Command

The Measure-Command cmdlet can be run to see how long it takes PowerShell to execute a command or script block. The cmdlet runs the script block internally and times how long it takes to execute, displaying the results after the script has executed.

It is a good way to get an apples-to-apples comparison of scripts.

Here we will run a simple get-content cmdlet across two remote computers. One computer, PC-01, is turned on and responsive. The other computer, PC-02, is turned off and unreachable remotely.

Figure 10.9 Measuring the response time for a remote query of an online vs an offline computer.



Two seconds might not seem like a lot. But if you have just 30 computers unavailable, that would add more than a minute to our total time to complete. If you have an enterprise with thousands or tens of thousands of systems, this can add considerable time to your request.

This is where the Test-Connection cmdlet can make our lives easier. Like a ping, Test-Connection sends out an Internet Control Message Protocol (ICMP) echo request packet to a destination or list of destinations. It is a quick way to see if a system is online and reachable before attempting to remotely access the system or its resources.

You can control the number of requests sent and received with the count parameter.

Ping vs Test-Connection

You may be wondering, *"Can't I already do all that with a ping? Why would I use Test-Connection?"* The reason is, that the test-connection can return a Boolean value as well based on the results of the echo request using the quiet parameter. With a traditional ping request, you would need to do additional conditional logic based on the response to determine if the ping was a success or a failure. This function is built into the Test-Connection cmdlet and requires no additional conditional logic or overhead.

Figure 10.10 Example of pinging with PowerShell

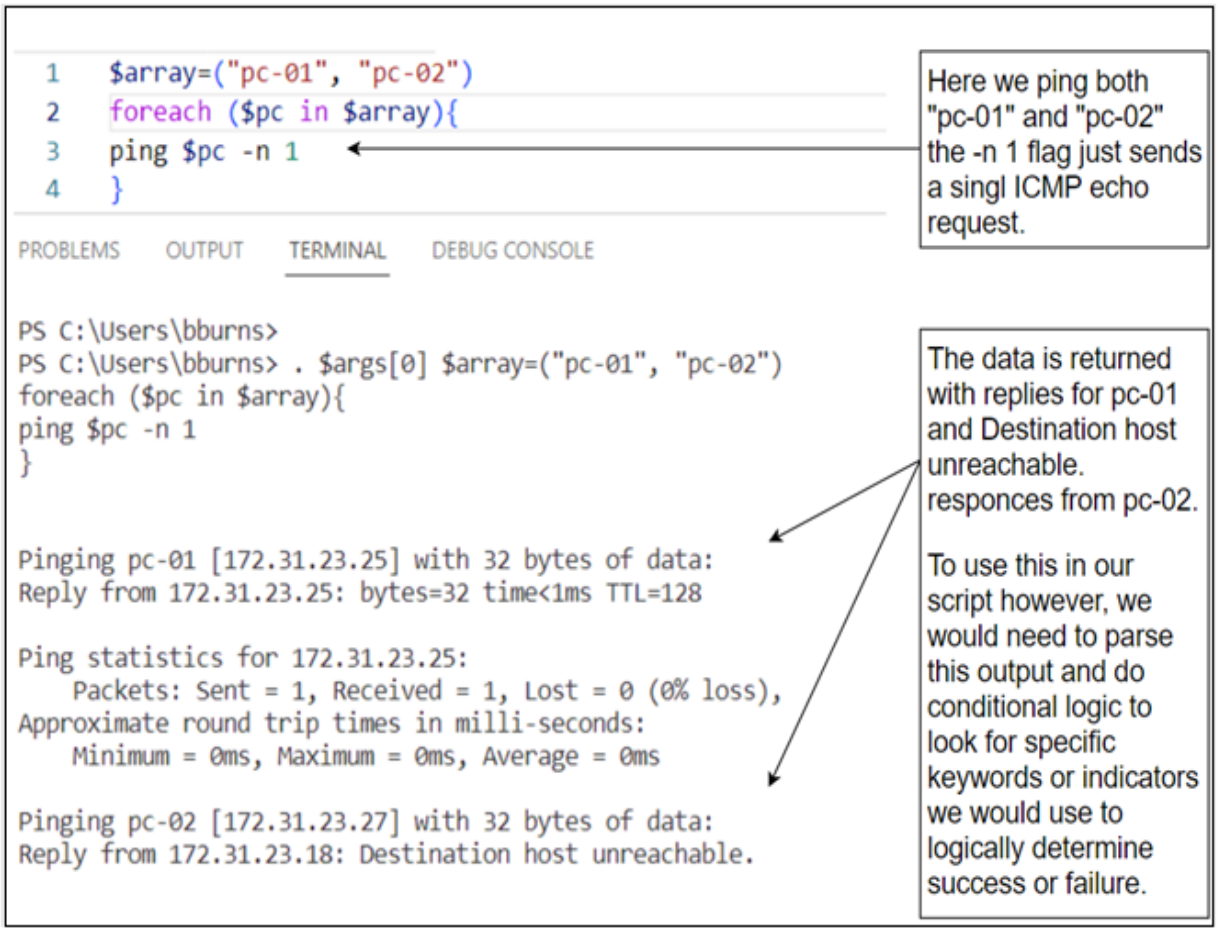
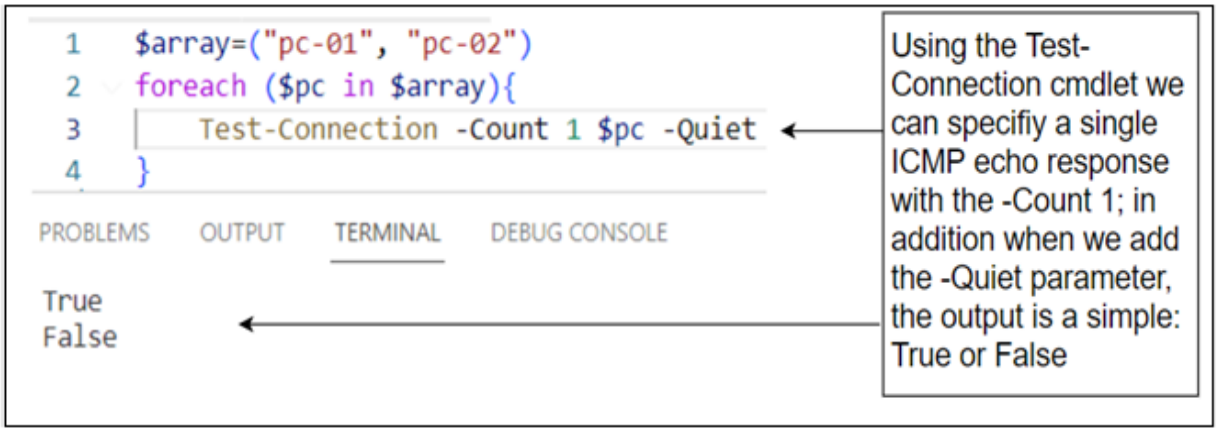


Figure 10.11 Using the Test-Connection -Quiet to get a True or False response.



10.3.1 The structure of the Test-Connection cmdlet

The syntax for the Test-Connection cmdlet includes several useful parameters:

- Count: Controls the number of ICMP echo requests sent
- TargetName: This can be one or more targets you wish to ping.
- Quiet: This causes the command to return only a Boolean value indicating success or failure.
- Traceroute: shows you all of the hops (or connections) your request is filtered through before it reaches its end destination.

For example, the following statement will send a single ICMP echo request to your host computer, also known as a loopback test.

```
Test-Connection -count 1 -TargetName $env:COMPUTERNAME
```

When we insert a Test-Connection into the same code we ran in Figure 10.9, we see that our time to execute is improved.

Figure 10.12 Use of a Test-Connection can improve script speeds by avoiding having to timeout commands.

10.3.2 How do we use this in our Tiny PowerShell Script?

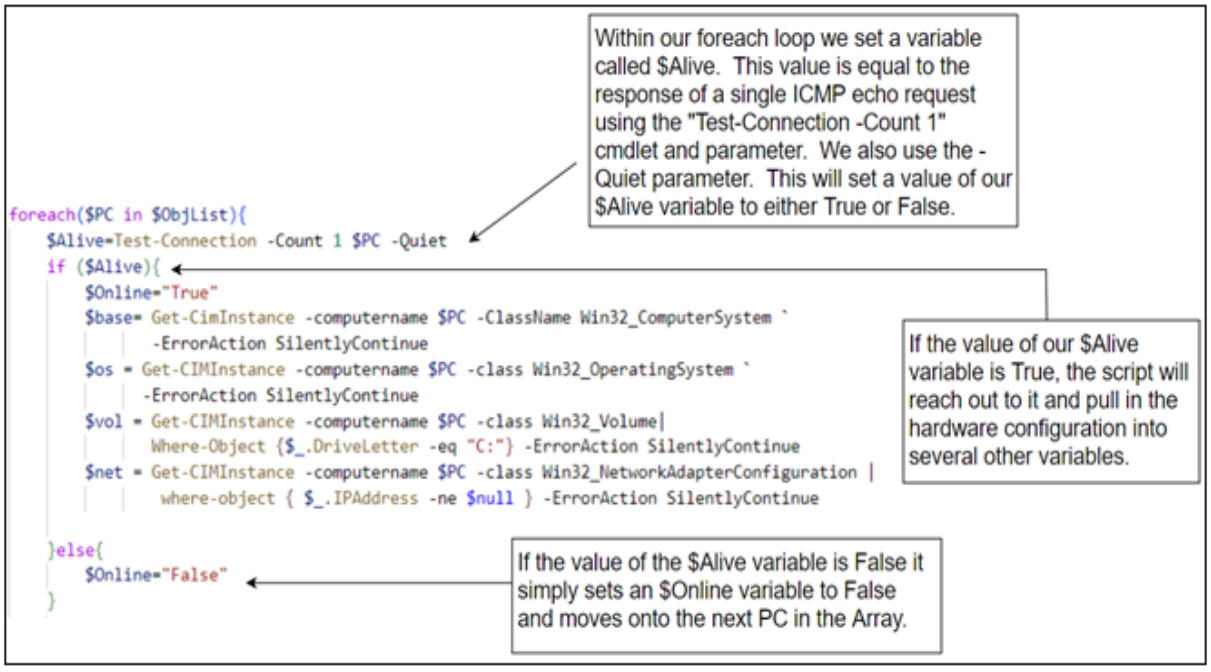
After we have established the list of computers we are going to query in our script, we then use a foreach loop to query each one for its hardware properties. However, because we don't want to waste time querying PCs that will not be able to answer (especially since we need to query each box

more than once), we first send a single ICMP echo request to the computer with the Test-Connection cmdlet to see if it is responsive.

We use the -Quiet parameter of the cmdlet to return only a True response if it is responsive or a False if it is unreachable. We then set an if statement to check the value of this response and, if the value is True, we query the system for its hardware.

We could move on without doing anything if the ICMP echo request is unreachable. Instead, we include the "Else" part of this script, where, if the value is false, we simply set a variable we call $Online to False before moving to the next computer in the list. We include this step because we will use this as a value in our report. When we deliver the report, it forestalls a lot of questions as to why fields are blank if the reader can immediately see that the system was not online when the query was run.

Figure 10.13 Illustrating how we use the Test-Connection cmdlet in our Tiny PowerShell Project.



10.4 Get-CimInstance

Computers store significant amounts of data, and a significant amount of this data is about the computer itself. This includes file permissions, registries, and information about its hardware, which is perfect for us! Accessing this data is easy and can be done remotely using PowerShell with the Get-CimInstance cmdlet.

CIM, which stands for Common Information Model, is an object-oriented, open-source standard for accessing and managing all kinds of computer, network, services, and application information. Microsoft has used this as the base of its Windows Management Instrumentation (WMI) since 1996. However, with the change from a single platform (Microsoft only) to multi-platform (Microsoft, MacOS, Linux…) support seen with PowerShell Core and beyond, CIM is the way to access this treasure trove of information.

When accessing the data within the CIM structure it's important to know how it is grouped. You won't, for example, be able to access the remote system's IP address from the same location as its Operating System version. This information is organized into data structures known as Classes.

10.4.1 Get-CimClass

Classes are the way that the data is structured. This keeps things that are alike grouped so you don't need to parse through the thousands of things your computer is keeping track of to find the thing you need. However, because of this grouping, you must know which class you need to query to get the data you want.

You can get a list of the available CIM classes your system knows about by running the Get_CimClass cmdlet.

```
Get-CimClass
```

After running that command you start to understand what I mean when I say your computer stores significant amounts of data about itself! It's probably an interesting use of your time to spend a little time digging around these classes to see what kinds of hidden information gems you might discover.

In general, it's probably a good idea to restrict your CIM Class search to Win32 as well. These are the classes, in general, that hold the most valuable information for System Administrators.

```
Get-CimClass |Where Get-CimClass|Where-Object {$_. CimClassName
-match "Win32"}
```

But knowing which Classes are available on our system is only part of the battle. We still need to know how to access this data. This is where the Get-CimInstance cmdlet comes in.

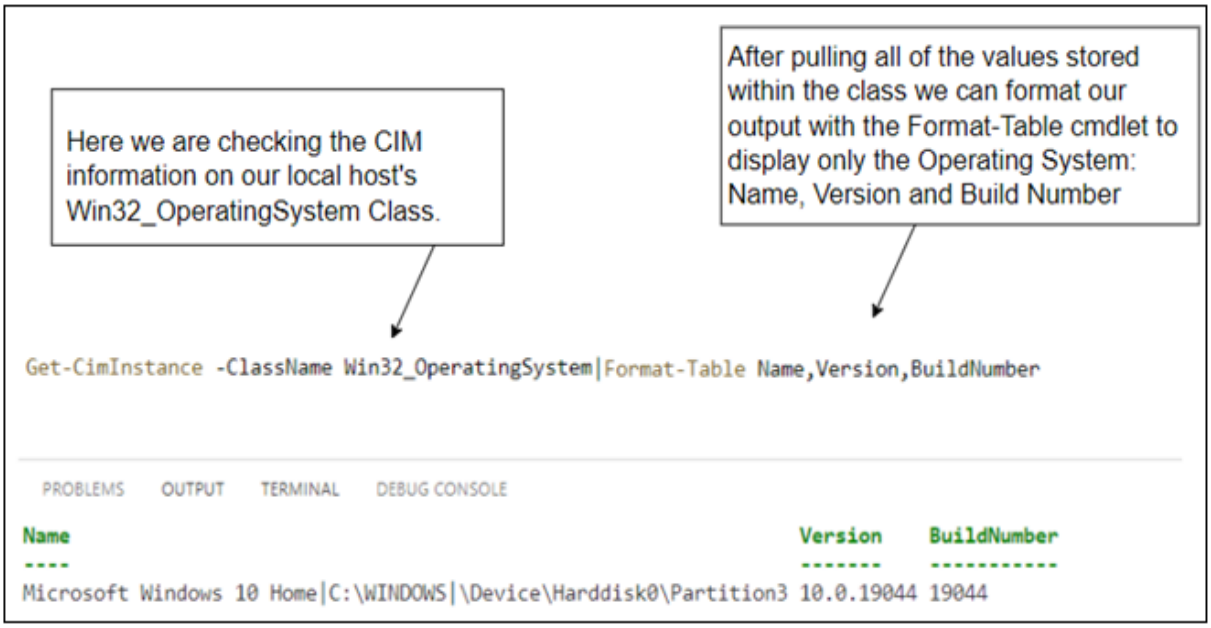
10.4.2 Get-CimInstance cmdlet structure

There are several parameters used with the Get-CimInstance cmdlet:

- ClassName: Once we know the CIM Class we want to look at we need to supply this value to the ClassName parameter.
- ComputerName: The ComputerName parameter allows us to remotely target remote computer CIM information. You can list one or more ComputerName values with the cmdlet.
- Property: The Property parameter allows us to either filter the list of returned values to those we care about specifically like the name, OS, or IPAddress. The cmdlet returns everything in the class by default

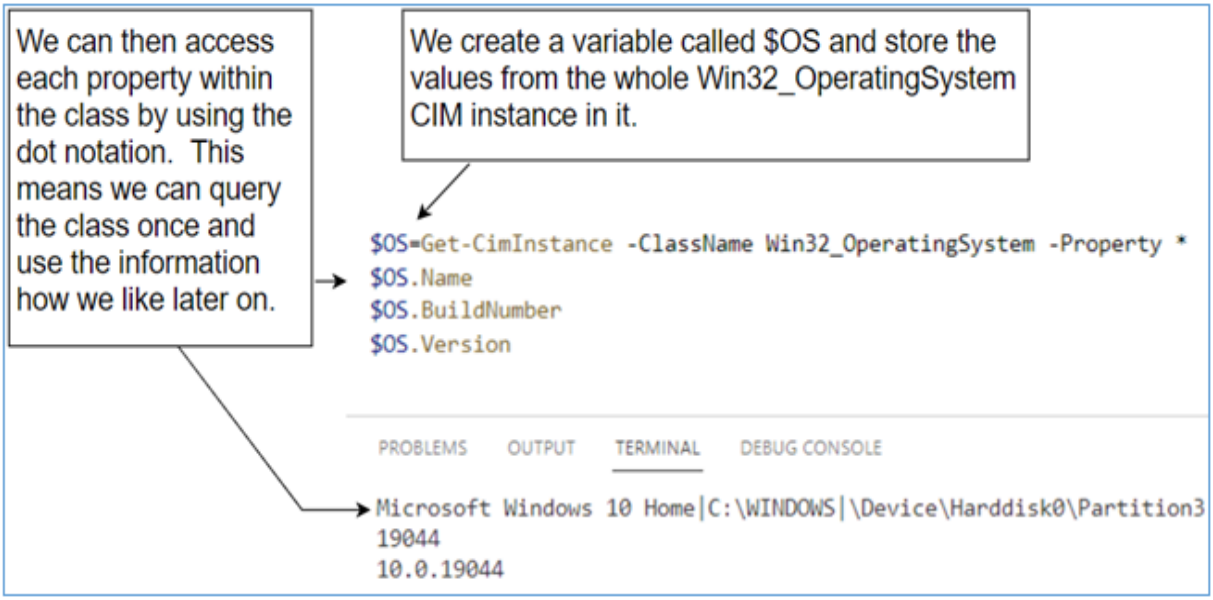
```
Get-CimInstance -ClassName Win32_OperatingSystem -Property
Version|Format-Table

Get-CimInstance -ClassName Win32_OperatingSystem|Format-Table
```

Figure 10.14 Querying system's operating system, version, and build with CIM classes.



You can also use dot notation to pull a specific property stored within a CIM Class. This allows us to pull the full class, store it as a variable, and then use the Dot notation to access specific aspects of the Class without having to constantly re-query the CIM Instance.

Figure 10.15 Saving the CIM Class as a variable and using Dot notation to access the properties.

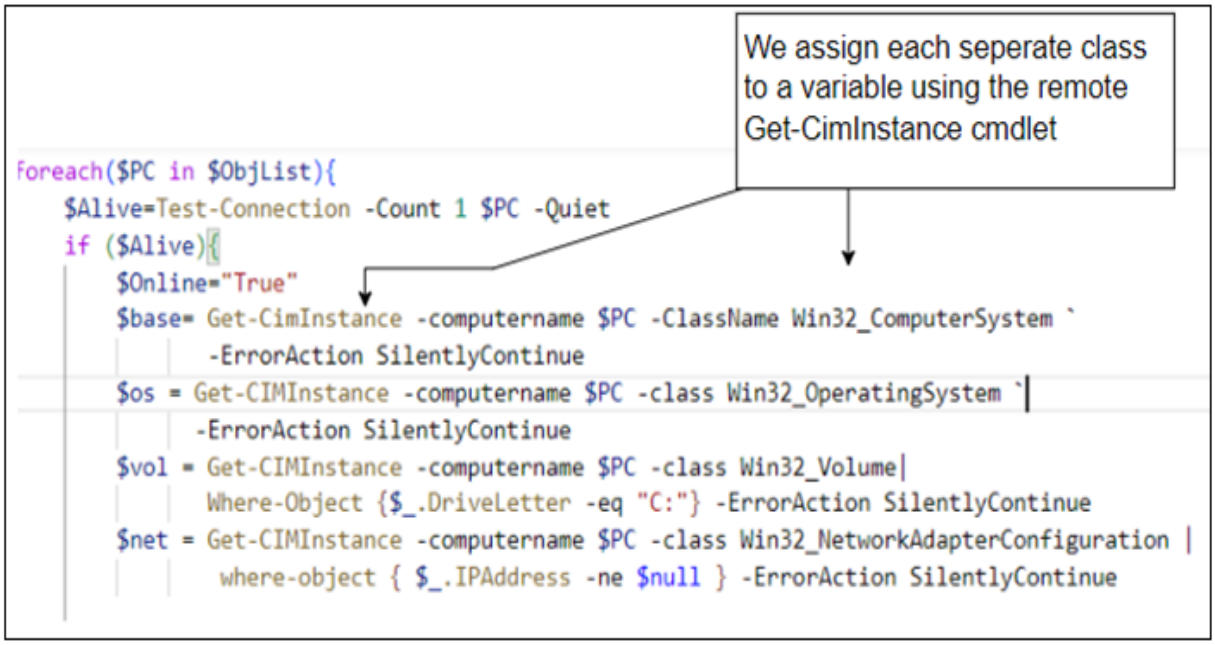


10.4.3 How do we use this in our Tiny PowerShell Script?

Once we've determined the computer is reachable on the network with the Test-Connection cmdlet, we then query the remote PC via the Get-CimInstance cmdlet, storing the class info from:

- Win32_ComputerSystem into a variable called $base
- Win32_OperatingSystem into a variable called $os
- Win32_Volume (for the C drive) into a variable called $vol
- Win32_NetworkAdapterConfiguration into a variable called $net

Figure 10.16 Assigning Get-CimInstance return values to variables in our Tiny PowerShell Script



Once we have the values for the four CIM instances stored in variables, we pull specific parts of the CIM Class out into other variables for our Hashtable (You'll learn about Hashtables in the next section). Using those four CIM Classes we can retrieve the values for the:

- Operating System
- Operating System Version
- Operating System Architecture

- Serial Number
- Organization
- Domain
- RAM
- IP Address IPv4 and IPv6
- Subnet Mask
- MAC Address
- HDD Disk Capacity
- HDD Disk Free Space

Figure 10.17 Pulling specific information from the CIM Class in our Tiny PowerShell Script.

```
$DeviceInfo= @{
    Online=$Online
    SystemName=$PC
    OperatingSystem=$os.name.split("|")[0]
    SerialNumber= $os.SerialNumber
    Version=$os.Version
    Architecture=$os.OSArchitecture
    Organization=$os.Organization
    Domain=$base.Domain
    RAM_GB=$base.TotalPhysicalMemory/1GB
    IPAddress=($net.IPAddress -join (", "))
    Subnet=($net.IPSubnet -join (", "))
    MACAddress=$net.MACAddress
    DiskCapacity_GB= $vol.Capacity/1GB
    FreeCapacity_GB=$vol.FreeSpace/1GB
}
```

10.5 Hashtables

A hashtable is simply a data storage structure like a variable or array. Each data item is stored in a key/value pair, sometimes referred to as a Dictionary value. A key/value pair, quite simply, is a unique string paired with another value. If we wanted to create a pairing of presidents with their vice presidents our key/value pairs might look something like this:

Vice Presidents

“George Washington” “John Adams”

“John Adams” “Thomas Jefferson”

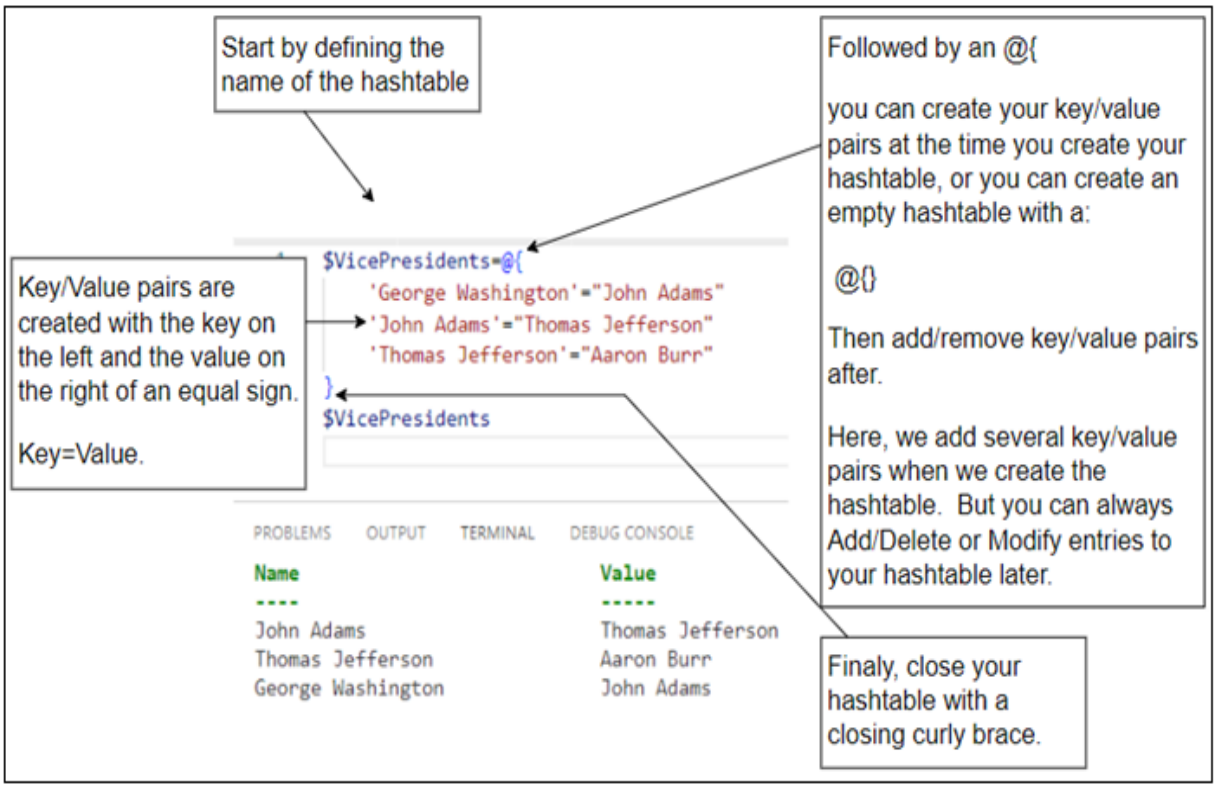
“Thomas Jefferson” “Aaron Burr”

10.5.1 Hashtable structure

To create a hashtable you first define the name of the hashtable with a leading $ and set it equal to an @{}. This is a similar structure to an array, where you create an array name followed by an @(). This is not the only similarity we’ll see between hashtables and arrays, but we’ll explore some differences as well. Here is how we would create and display our vice presidents hashtable:

```
$VicePresidents=@{
    'George Washington'="John Adams"
    'John Adams'="Thomas Jefferson"
    'Thomas Jefferson'="Aaron Burr"
}
$VicePresidents
```

Figure 10.18 Structure of a hashtable.



10.5.2 Benefits of using a hashtable in PowerShell

One thing you may notice about the vice presidents hashtable is that the data is not stored in the same order that we supplied it. Our first key/value pair was 'George Washington' and "John Adams". But when we query the hashtable, the first entry is 'John Adams' "Thomas Jefferson". This is because the hashtable decides on the placement of the key/value entry into the backend array with its own algorithm. Arranging the data with this algorithm makes finding a specific key/value pair very fast.

Let's say we wanted to know Thomas Jefferson's vice president. We could simply query the array with the key in square brackets.

```
$VicePresidents['Thomas Jefferson']
```

Once again, the syntax is very similar to accessing an array. But unlike when accessing an array, we do not need to know which position the data was entered in beforehand. The hashtable's algorithm can find it for us much quicker than we could search an array.

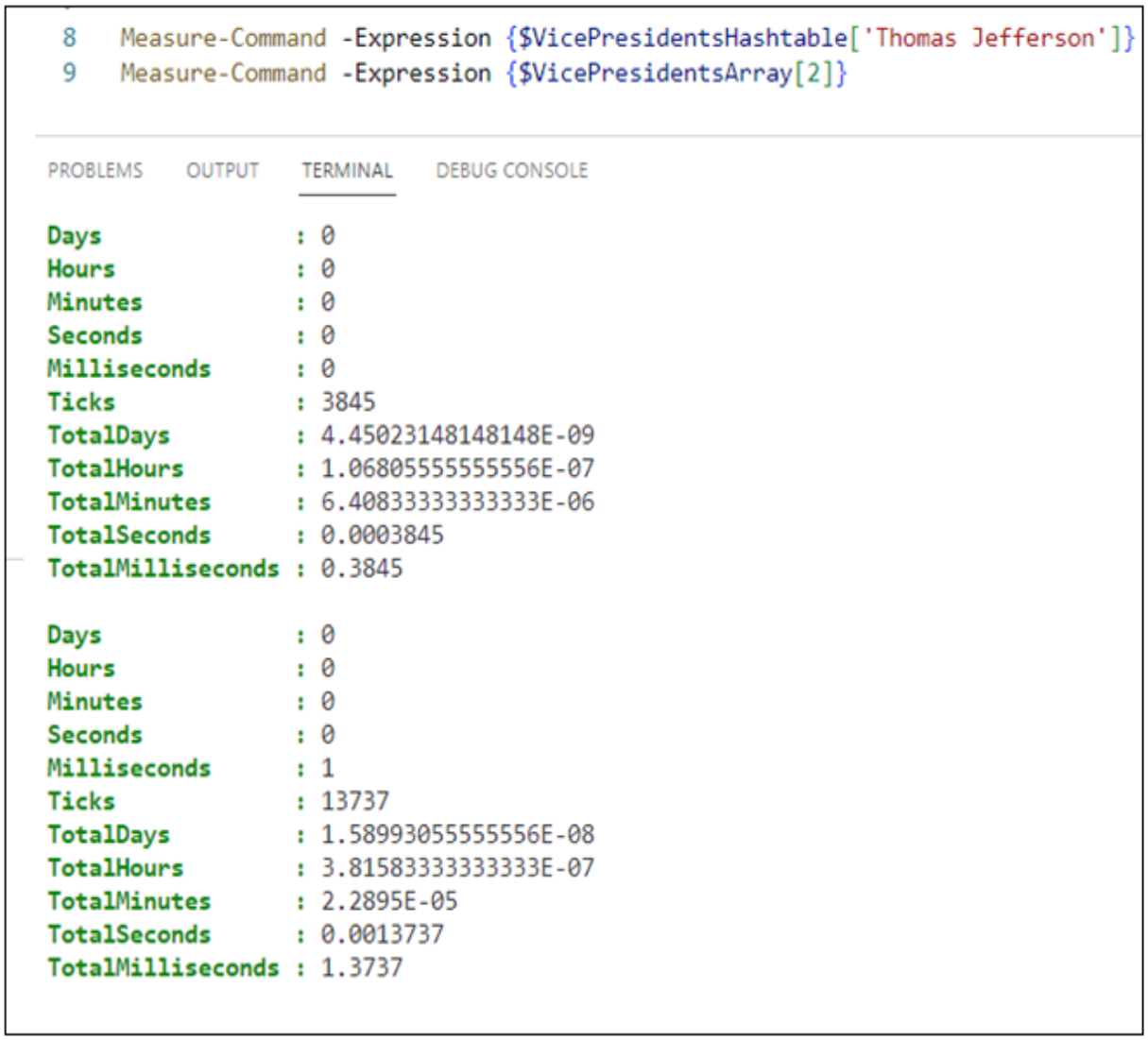
Let’s set up a hashtable and an array with this information:

```
$VicePresidentsHashtable=@{
    'George Washington'="John Adams"
    'John Adams'="Thomas Jefferson"
    'Thomas Jefferson'="Aaron Burr"
}
$VicePresidentsArray=@("John Adams", "Thomas Jefferson", "Aaron
Burr")
```

Using the Measure-Command we can see how long it takes to find Aaron Burr in the array.

```
Measure-Command -Expression {$VicePresidentsHashtable['Thomas
Jefferson']}
Measure-Command -Expression {$VicePresidentsArray[2]}
```

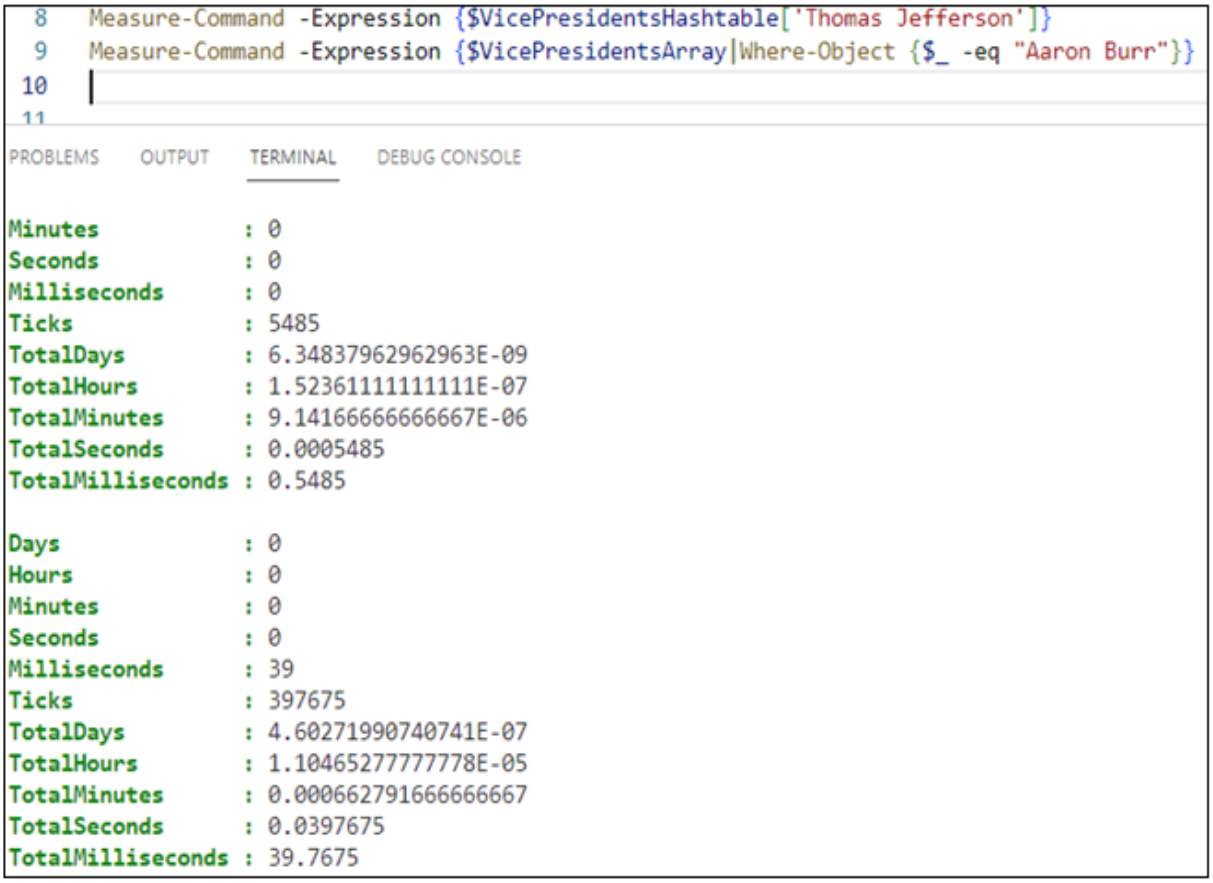
Figure 10.19 Comparing the speed of data access between an array and a hashtable.



This impact is even larger if you don't already know the position of the data you want to pull within the array. Let's say you wanted to pull Aaron Burr from both the hashtable and the array. To access the answer from the hashtable you just need to know the key but to access the answer from the array you'll need to look at all the entries and select the one you want with a Where-Object cmdlet.

```
Measure-Command -Expression {$VicePresidentsHashtable['Thomas
Jefferson']}
Measure-Command -Expression {$VicePresidentsArray|Where-Object
{$_ -eq "Aaron Burr"}}
```

Figure 10.20 Speed comparison between arrays and hashtables when the array's index value is unknown.



10.5.3 How do we use this in our Tiny PowerShell Project?

After we have pulled the CIM Class data from a PC we create a hashtable called $DeviceInfo. We then use the four variables containing the classes we stored to make key/value pairs for each hardware category we care about before exporting the whole table of keys and values to our CSV.

Figure 10.21 The hashtable $DeviceInfo used in our Tiny PowerShell Project.

```
$DeviceInfo= @{
    Online=$Online
    SystemName=$PC
    OperatingSystem=$os.name.split("|")[0]
    SerialNumber= $os.SerialNumber
    Version=$os.Version
    Architecture=$os.OSArchitecture
    Organization=$os.Organization
    Domain=$base.Domain
    RAM_GB=$base.TotalPhysicalMemory/1GB
    IPAddress=($net.IPAddress -join (", "))
    Subnet=($net.IPSubnet -join (", "))
    MACAddress=$net.MACAddress
    DiskCapacity_GB= $vol.Capacity/1GB
    FreeCapacity_GB=$vol.FreeSpace/1GB
}
```

10.6 Export-CSV

In previous chapters, we already explored some of the powerful uses of a CSV (Comma-Separated Values) data format. For instance, this data type can be read by text editors and with Microsoft Excel, which makes it perfect for reports. But in those previous chapters, we were Importing-CSV files for use within our script. Now that we have this wonderfully structured data from our hashtable, you might wonder, “How do we save this data while retaining its structure?” That is where the Export-CSV cmdlet comes in.

10.6.1 Structure of the Export-CSV cmdlet

We’re pretty comfortable by now writing data out, to either the output, the host, or even a text file. Fortunately, the Export-CSV cmdlet is pretty straightforward and there are only a few options that we need to pay much attention to.

- -Path: This is the location of the file you’d like to save to. This parameter is the only one of the parameters we’ll be covering that is required.

- -Append: Adds data to the bottom of the file, instead of overwriting it.
- -Delimiter: The default is a comma, but this can be changed.

10.6.2 How do we use this in our Tiny PowerShell Script?

After we have constructed the hashtable from our CIM Classes, we call the contents of the hashfile, then pipe the contents into the Export-CSV cmdlet using the -path parameter to point back to the $SavePath parameter of our script. We also use the -append flag to add the current hashtable key/values to the CSV. This is because we are executing the Get-CimInstance and the creation of the hashtable within the foreach loop. We want to make sure we record the data for each PC's CIM Classes and Hashtables to the report, then move on to the next, ensuring we pull the appropriate hardware information for each PC in our $Objlist. Also, the -Append flag is very important. Without this flag, the most recent data in the loop is written to the file, the next item in the list then overwrites this file and the next one overwrites it. With the -Append flag each iteration of the loop writes its data and the next one appends its data to the list.

Figure 10.22 using the Export-CSV cmdlet in our Tiny PowerShell Project

```
$DeviceInfo| Export-CSV $SavePath -Append
```

However, because the path parameter is looking for a .csv file, we have to make sure that the $SavePath includes the .csv file type. Thus, before writing to our csv, we put in a guard to ensure that the $SavePath meets our formatting needs.

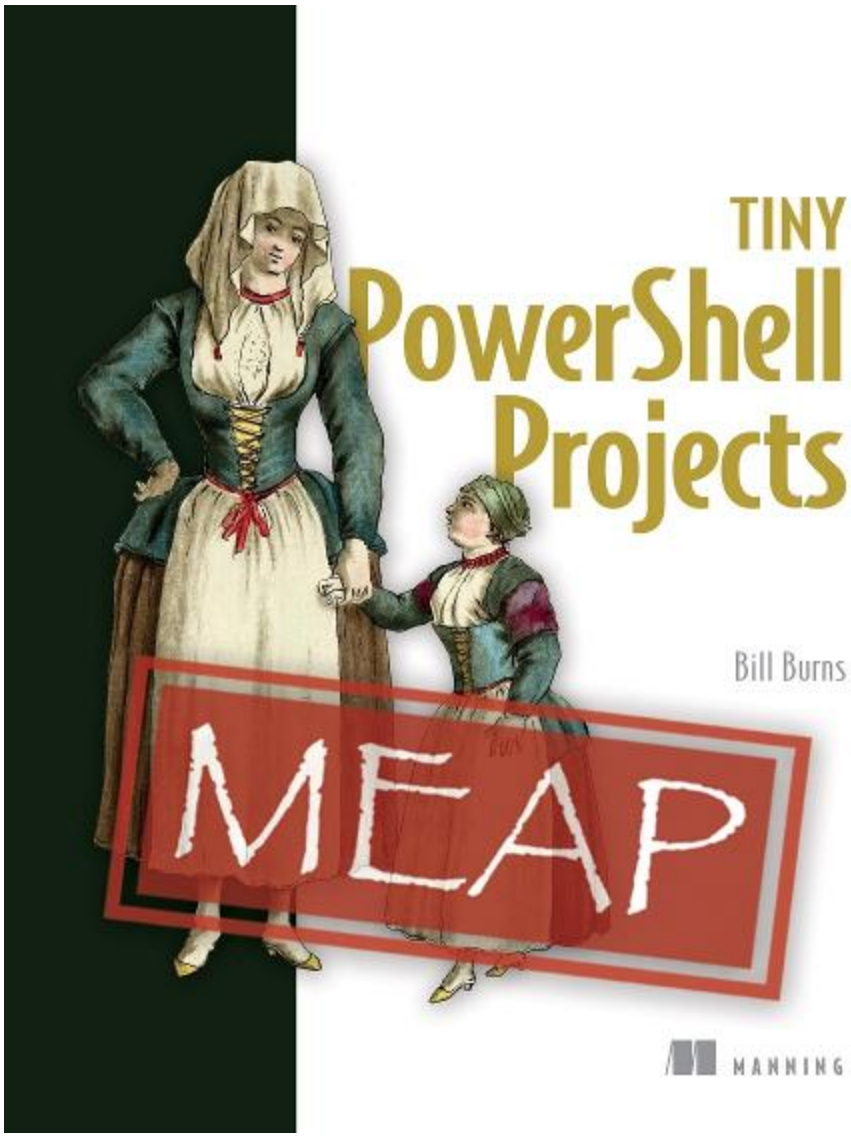
We set up a while loop that checks the value of the $SavePath and, while the value within this variable does not include a .csv extension, it will prompt and re-prompt our user to re-enter the save path with a file that ends in the .csv file extension.

Figure 10.23 Guard statement to make sure the $SavePath variable includes a .csv extension.

```
while ($SavePath -notlike "*.csv"){
    Write-Error "The save path must end in a csv file name `n Example: c:\temp\report.csv"
    $SavePath=Read-Host "Enter the full path and name of the csv to save the data to"
}
```

10.7 Summary

- Switch allows you to compare the value of a variable against several options and execute the code associated with the first match without using multiple if/else statements.
- Using Read-Host and Switch statements, you can make simple menus that allow your user more control over what can execute within your script.
- The Test-Connection cmdlet allows you to send out ICMP echo requests to remote computers to check to see if they are online before attempting to remotely access them, improving the efficiency of your scripts in most cases.
- The Test-Connection cmdlet is similar to the ping command, but if you use the -quiet parameter it will return a Boolean value instead of response metrics, making conditional logic based upon the responses much faster and easier.
- Computers store massive amounts of data, and much of this can be accessed remotely with the Get-CimInstance cmdlet.
- The CIM (Common Information Model) data is broken up into CIM Classes; these classes can be displayed with the Get-CIMClass cmdlet.
- Hashtables are a quick and efficient data storage type used in PowerShell to store unique key/value pairs.
- These Key/Value pairs can be referenced by calling the name of the hashtable followed by the unique key name in square brackets.
- The Export-CSV cmdlet allows you to export data from PowerShell into a structured data file known as a Comma-Separated Values file. This data type can be read by text editors and with Microsoft Excel, which makes it perfect for reports.


TINY
PowerShell
Projects
Bill Burns
MEAP
MANNING