Global Knowledge®

Expert Reference Series of White Papers

Five Simple Symbols You Should Know to Unlock Your PowerShell Potential

Five Simple Symbols You Should Know to Unlock Your PowerShell Potential

Jeff Peters, MCSE

Introduction

Recently, I had been asked to automate the creation of over one thousand new user accounts in the Active Directory® domain of one of my clients. This is a yearly process that had been done manually in the past, and it had required dozens of hours by administrative and IT staff members to get it done. When I mentioned that PowerShell could easily automate the process and avoid many of the errors you encounter with data entry done by hand, it was an easy choice for them. There was, however, a complaint that came after I had finished the script, executed it, and created all the accounts without a single error.

My direct contact at the company was their IT director. He said the problem was with the script. He loved the results of the script, but when he looked at the PS1 file (PS1 is the file extension that most PowerShell scripts use), it may as well have been written in Greek. Even after reading the comments that I had included in the script, he admitted that he just couldn't follow the syntax. I spent the next fifteen minutes walking him through the script line by line explaining what was happening at each stage.

Many people in IT fall into this same predicament. Since using PowerShell often comprises a small portion of what you may be responsible for doing, you don't learn its intricacies. In most cases, people google what they are looking for and download a script to do what they need. With good reason, however, they are reticent to execute a script on their network without a clear idea of exactly what it does. The skill you need is the ability to translate PowerShell into plain English!

The Five Symbols

There are five symbols that you will see regularly in PowerShell scripts. If you have experience with other scripting or programming languages, they may even be familiar to you. The purpose of this white paper is not necessarily to teach you how to use these symbols. Instead, our goal is to allow you to read and recognize them so that you can understand their meaning when they are used in existing scripts.

The five symbols and their basic meanings are:

@	hash table
?	Where-Object
|	passes results of one command to the next command
$	indicates a variable
%	ForEach-Object

Some of these symbols are shortcuts used to condense a script; others are notations used to allow customized displays of information. The pipe (shift-backslash on your keyboard) allows you to join individual command-lets (cmdlets) together. They are all useful, but can easily lead to confusion due to their lack of obvious meaning when seen by an inexperienced reader.

Aliases

Most commands in PowerShell are written as cmdlets and normally follow a verb-noun format. An example would be "Get-Service" with *Get* being the verb and *Service* being the noun. They are relatively easy to read and comprehend. The confusion arises when people use aliases which are accepted abbreviations for many commands. The alias for Get-Service is "gsv" and it returns the same results as the cmdlet. Many of the five symbols I mentioned earlier also act as aliases, so understanding their meaning can help to decode many scripts.

To see a complete list of these aliases, you can access PowerShell's alias drive:

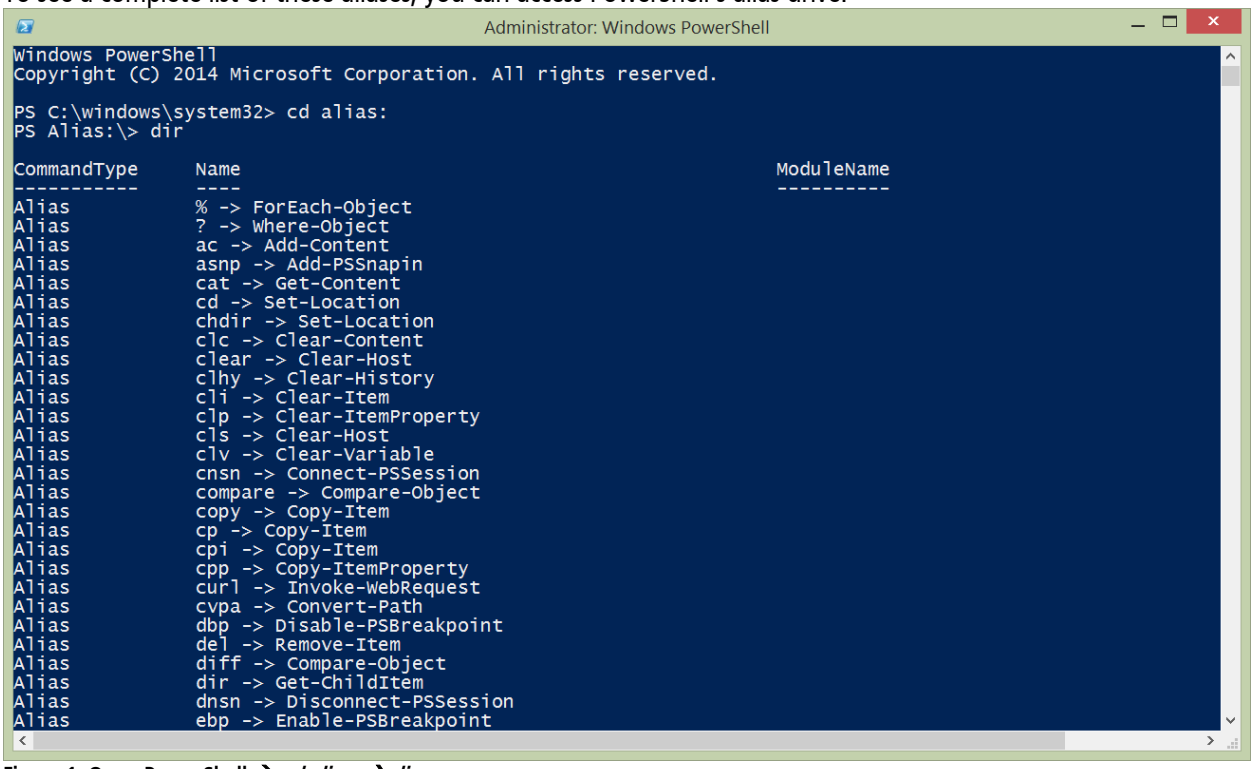

Figure 1. Open PowerShell → *cd alias:* → *dir*

PowerShell does allow you to add your own aliases, and they will be displayed in this list. Be aware, however, that the default behavior is that your aliases will only exist as long as you have this console open. Each time you open a new session, the aliases would have to be recreated.

Symbol 1

@ hash tables (aka custom columns)

Running a cmdlet such as Get-Process returns a good amount of obvious information about your current processes, including things such as the process names. There is another column, labeled VM that people may not recognize.

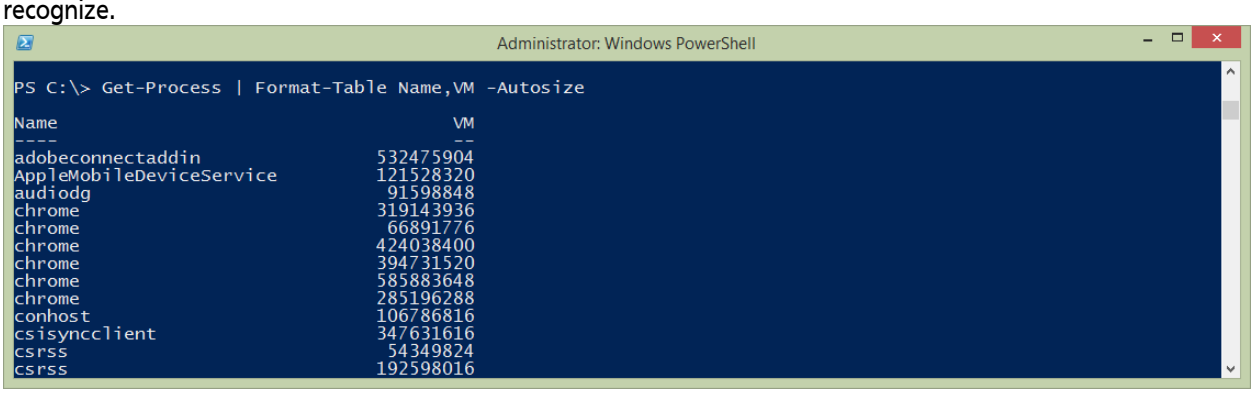

Figure 2. Results of Get-Process cmdlet that is formatted as a table and returns two columns

In this example, VM stands for Virtual Memory and the value is shown in bytes. To avoid confusion when displaying this type of information (or when preparing this data to send to a report), many PowerShell programmers create their own custom columns which display the information in an easier to understand format. While this practice makes the output easier to read, the script itself looks drastically more complex unless you are familiar with @ and the custom columns it is used to create.

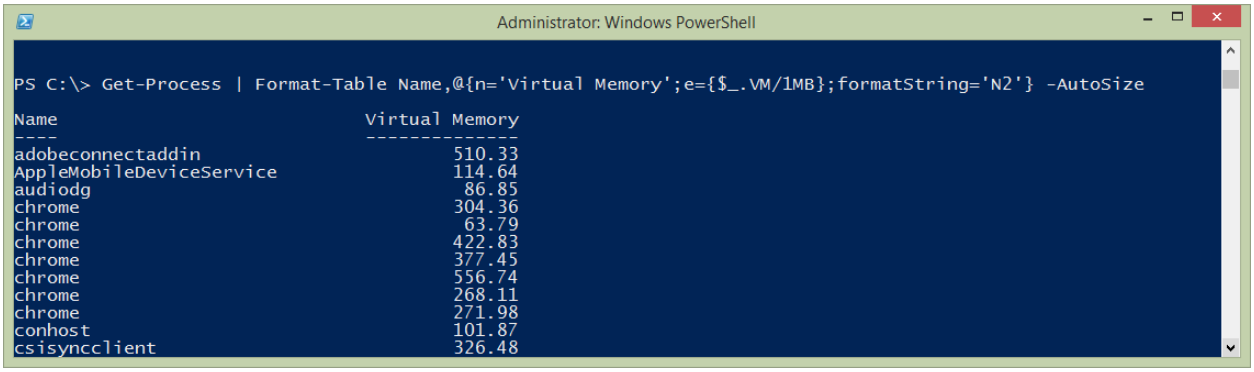

Figure 3. Using a hash table to produce an easier to understand result

The cmdlet used in figure 2 lists the two columns and formats them as a table. The displayed information is correct, but the VM column label may be unclear and the amount of virtual memory is shown in bytes. The cmdlet in figure 3 also lists the two columns and formats them as a table. The VM column, however, has been customized to display a header that reads "Virtual Memory" and the data is shown in megabytes.

The custom column is created with the following syntax:

@{n='Virtual Memory';e={$_.VM/1MB};formatString='N2'}

The @ is the indicator to PowerShell that it will be a custom column. The "n" sets the name or label that will be displayed at the top of the column and is encased in single quotes. The "e" (which is separated from the name by a semicolon) sets the expression, which determines the data that will be shown in the column. The optional "formatString" sets the format to a number with two digits to the right of the decimal point. If you translated this string into plain English, it would read: "Display a column that is labeled 'Virtual Memory'. The contents of

the column should be the contents of the VM column, but display it in MB by dividing the total number of bytes by 1MB (1024 bytes). Format it so that no more than 2 numbers to the right of the decimal are shown."

Be aware that both the name and expression fields are going to display *exactly* what you tell them to. In other words, if you misspell the name or screw up the expression, that is what will be shown on the screen.

Symbol 2

? alias for Where-Object

It is normal to want to look at a smaller subset of data when querying with PowerShell or any other tool. For example, maybe I don't want to look at all of the services that a Get-Service cmdlet would return; I just want the services that are stopped. The statement used to make this happen is Where-Object. It acts as a filter which only returns objects that evaluate to true against the test statement that follows it.

When reading a script that contains a Where-Object cmdlet, it is pretty easy to figure out what's going on. But, of course, that cmdlet rarely shows up in most scripts due to the built-in ability to shorten the syntax of that cmdlet. You can use the word *where* by itself, but that isn't too difficult to figure out its meaning. The trouble comes when the alias for Where-Object is shortened to a single "?".

Compare the following two queries:

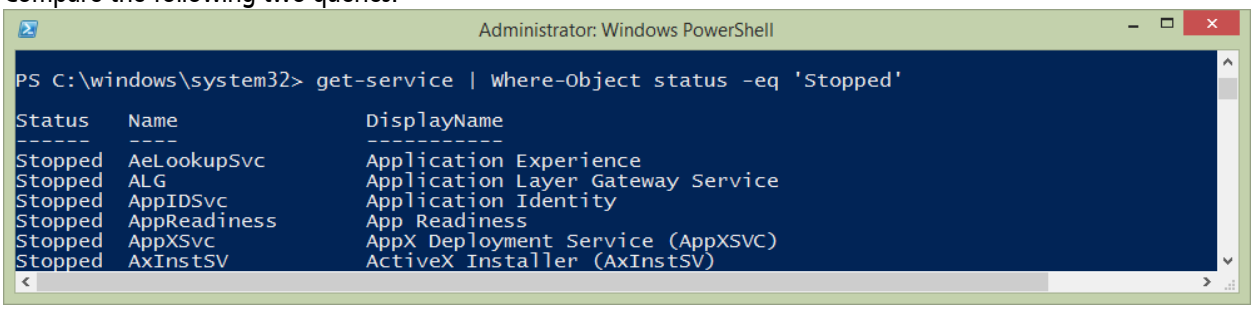

Figure 4. Get-Service | Where-Object status –eq 'Stopped'

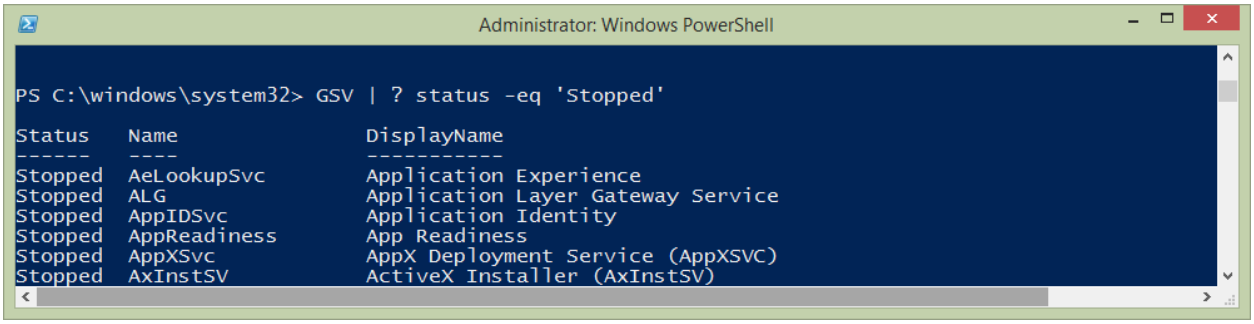

Figure 5. GSV | ? status –eq 'Stopped'

The two queries do EXACTLY the same thing. The only differences stem from the use of aliases which certainly shorten your typing time as the author, but end up mystifying the average user who tries to decipher the command. GSV is the shortened form of Get-Service and ? is a built-in alias for the Where-Object command.

The "?" is regularly used in scripts, but since there is no obvious relationship between it and the replaced Where-Object cmdlet, people who are unaware of its meaning will find themselves at a loss as to what it means. Once the meaning is revealed, its use becomes obvious. Translating the cmdlet to plain English would result in: "Get a list of services from the local machine, but only show those services that are currently stopped."

**It is also worth noting the "–eq" that is used in the syntax above. When comparing items in PowerShell, you use a special set of comparison operators.

–eq equal
–ne not equal
–lt less than
–le less than or equal to
–gt greater than
–ge greater than or equal to
–like comparison using wildcards
(**prefacing any of these commands with a "c" makes them case-sensitive. –ceq equal and case-sensitive)

Use of an equals sign or similar symbol in this type of situation will almost always result in an error and red filling your screen.

Symbol 3

| (the "pipe") allows you to pass the results of one cmdlet to another cmdlet for processing

The pipe is probably the most used and the most powerful of the five PowerShell symbols we are examining. (If you are unfamiliar with the pipe, you can find it above the backslash on your keyboard.) Its purpose is simple. Whenever a cmdlet is executed, a result set of objects known as a collection is generated. If you would like to process that collection further, you pass the collection on to another cmdlet via the pipe. Processing that collection further might consist of filtering services, then piping them to be sorted, then piping them again to be fed into a comma-separated values (CSV) file. You are not limited to a single pipe action, either. The pipeline, as it is called, can continue on getting more precise and completing more actions as it passes through pipe after pipe.

You probably noticed the use of the pipe in the previous two examples. In the Get-Process command shown in figures 2 and 3, you will see that once the list of processes is gathered, they are formatted by being piped to a second cmdlet. In the Get-Service example shown in figures 4 and 5, once the services are collected, they are filtered by being piped to a second cmdlet.

Here's a different example. You are asked to disable a user account, and you would like to use PowerShell to do so. (I sound like an exam question . . .) If you happen to know the exact username of the person, you can use that cmdlet to disable them immediately. For the sake of example, let's say that you only know the guy's first name: Don. Now what? Instead of giving up and opening Active Directory® Users and Computers to search for him, we can filter using PowerShell.

We start with a simple query of Active Directory®. Get-ADUser requires a filter of some sort, so we use an * to say we will start by looking at everyone.

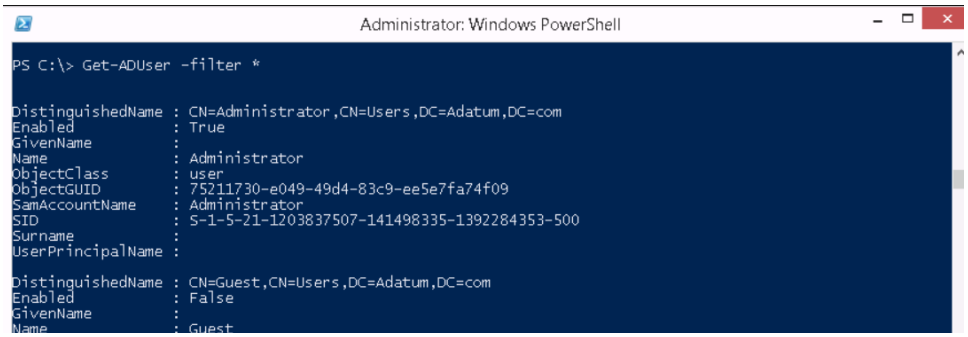

Figure 6. Get-ADUser –filter *

While this is a good start, we are given a list of every user account in our domain. We need to narrow the list. Making use of our knowledge of the ? (Where-Object) cmdlet, we can do this easily by piping the initial list of all Active Directory® users into a filter to constrain the results based on Don's given name. (FYI givenName is the technical label for the first name field in Active Directory®.)

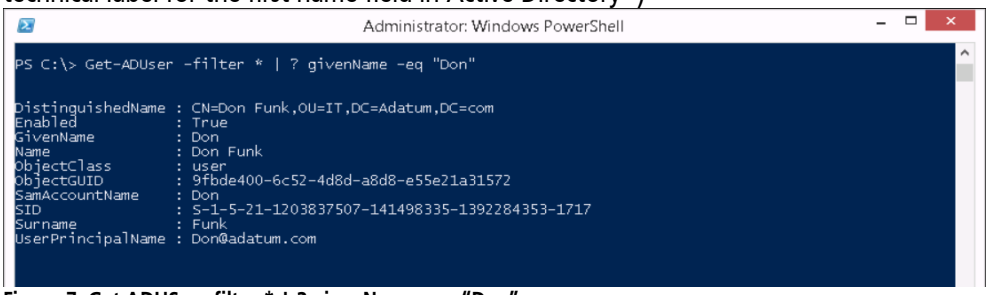

Figure 7. Get-ADUSer –filter * | ? givenName –eq "Don"

Our pipeline returns exactly the results we need. Of course, if there was more than one Don, we could further refine our filter based on other fields. We can see his user account has an Enabled field that is currently True. If we are successful, we will set that to False.

The final step to accomplish that is to use this result by passing it through an additional pipe to execute the Disable-ADAccount cmdlet against it.

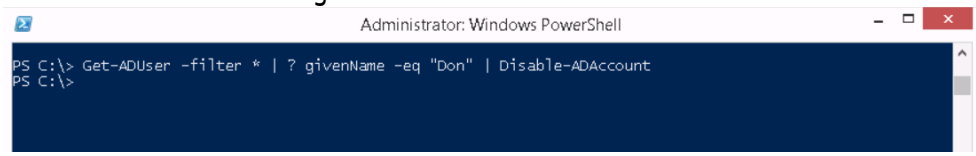

Figure 8. Get-ADUser –filter * | ? givenName –eq "Don" | Disable–ADAccount

This is where you had better check your results from the filter *before* you execute. Any accounts that are passed through the pipeline will be disabled. If you leave out the filter entirely, for example, ALL ACCOUNTS will be disabled. PowerShell truly is powerful, which can lead to dangerous results if used improperly.

Symbol 4

$ (variables)

Writing scripts can automate many tasks, but each script often serves a very specific function. Adding parameters and variables to your scripts can make them drastically more flexible. This is where the $ comes in. Whenever you see a $ preceding a value, it signifies that you would like to replace that value with the contents of the variable it represents.

Let me clarify that. Variables are storage containers. When you use a variable in your script, you are telling the script to replace the variable with its contents. First you set the contents of the variable by setting it equal to something. This process will also create the variable if it didn't already exist. If you have a variable name *var1*, then you would access its contents by representing it as *$var1*. In figure 9, the variable is created and its contents are set. Then it is accessed by simply typing $var1. The contents are then revealed.

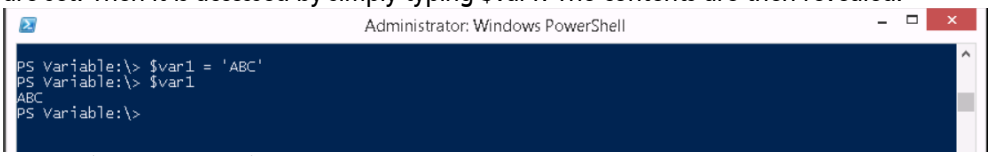

Figure 9. $var1 = 'ABC' → $var

You can examine all of the current variables on your system by accessing PowerShell's variable drive. (Don't forget the colon when changing directories to see it.)

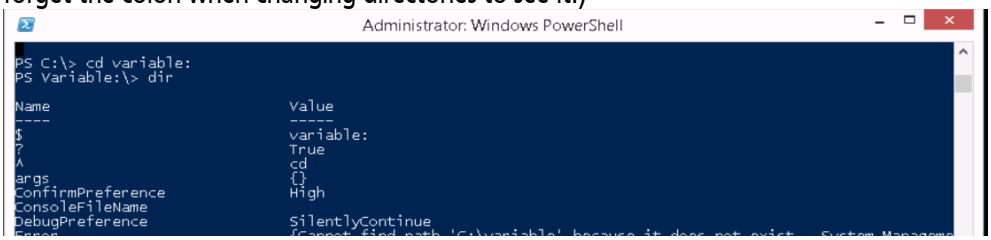

Figure 10. cd variable: → dir

Symbol 5

% alias for ForEach-Object

The % symbol represents the ForEach-Object cmdlet. It allows you to perform an action against multiple objects when that action normally only works on one object at a time. It creates a loop where the action is performed against the first object in the list, and, upon completion, loops back up to perform it on the next object in the list until all objects in the list have been addressed.

In figure 11, the script attempts to obtain a list all of the services followed by piping that list into a command to display the services in different colors, depending on whether they are running or stopped. Obviously, this results in an error. The key to fixing this issue is in the fourth line of the error where it reads: "Expressions are only allowed as the first element of a pipeline." In other words, PowerShell won't use an expression, such as the IF statement used to color the services, unless it is the very first item in the pipeline.

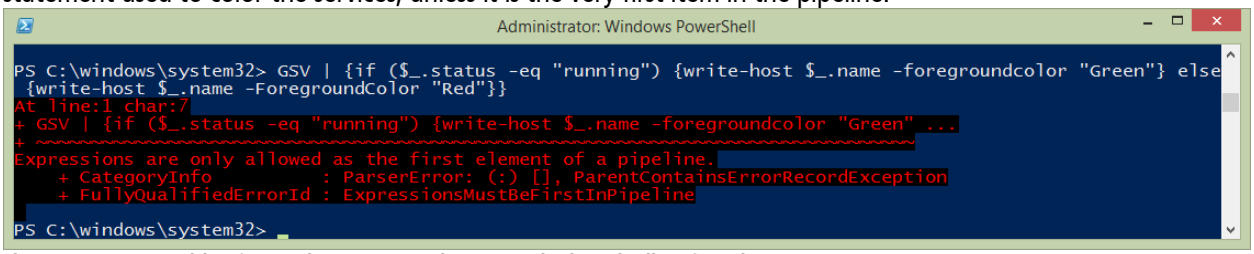

Figure 11. Error resulting from using an expression to manipulate the list of services

To get around this limitation, we use a ForEach-Object statement. Again, this is a very common cmdlet, and is represented with an alias of "%". Obviously, this symbol has no direct correlation to the cmdlet, so, once again, it can be very difficult to figure out what is occurring in a script when it appears.

The straightforward reasoning as to why this command fails is that PowerShell has one expression to apply against a full list of services. It seems obvious that it should apply the expression against each iteration of a service, but we have to tell PowerShell explicitly to do so. This is done with the %.

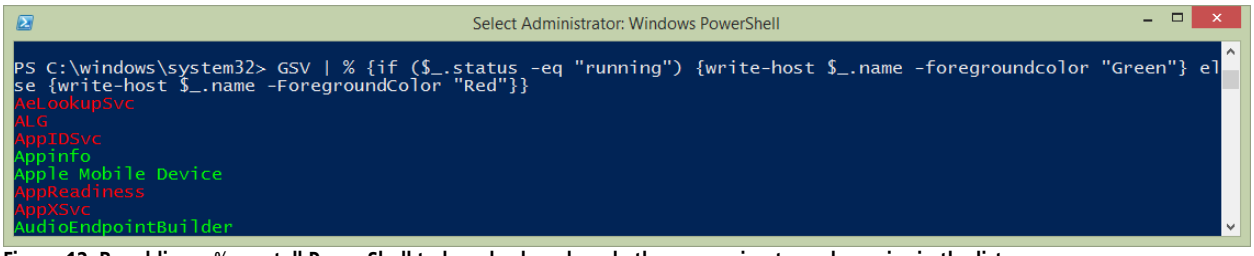

Figure 12. By adding a %, we tell PowerShell to loop back and apply the expression to each service in the list.

The ForEach-Object statement that the % represents tells PowerShell that when it receives the list of services through the pipeline, it should apply the expression against the first object, AeLookupSvc. It determines that the service is not running, so it changes its color to red. Then it looks at each following service individually to make the same decision. If it is running, it's displayed in green, or, if it is stopped, it's displayed in red. We are allowing PowerShell to loop through the list one service at a time. That way the application of the expression is only applied against a single object at a time.

**The only remaining, unexplained object is the $_. It represents the current item being passed through the loop.

Conclusion

Five symbols are regularly used in PowerShell: @, ?, |, $, and %. Unless you know their meanings, they can obscure the purpose and actions of a PowerShell script. There will always be new cmdlets to decipher. Each new operating system release and application version will include new and useful cmdlets along with new parameters for existing cmdlets. You can't memorize them all, but with the basic knowledge of these particular symbols, you should be well equipped to follow the logic of many scripts.

Learn More

Learn more about how you can improve productivity, enhance efficiency, and sharpen your competitive edge through training.

Automating Administration with Windows PowerShell (M10961)

PowerShell Essentials

Windows PowerShell Scripting and Toolmaking (M55039)

Visit **www.globalknowledge.com** or call **1-800-COURSES (1-800-268-7737)** to speak with a Global Knowledge training advisor.

About the Author

Jeff Peters has over twenty years of system administration experience, and has been Microsoft certified since getting his MCSE in NT 4.0. He works primarily with AD, SQL, and System Center products, and splits his time between consulting and training through his company, Azen Tech. Jeff resides in Metuchen, NJ, with his wife and two sons, who all roll their eyes when he gets overly excited about technology.