Mastering Windows PowerShell Scripting

Master the art of automating and managing your Windows environment using PowerShell

Foreword by Jon Stevens, Chief Information Officer, CDW

Brenton J.W. Blawat
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Functions, Switches, and Loops Structures'
- A synopsis of the book’s content
- More information on Mastering Windows PowerShell Scripting

About the Author

**Brenton J.W. Blawat** is an entrepreneur, strategic technical advisor, author, and senior consultant, who has a passion for the procurement of technology in profit-based organizations. He is business-centric and technology-minded. Brenton has many years of experience in bridging the gap between technical staff and decision-makers in several organizations. He takes pride in his ability to effectively communicate with a diverse audience and provide strategic direction for large and small organizations alike.


Brenton currently works at CDW as a senior consulting engineer in strategic solutions and services. CDW is a leading multibrand technology solutions provider in the fields of business, government, education, and healthcare. A Fortune 500 company, it was founded in 1984 and employs approximately 7,200 coworkers. In 2014, the company generated net sales of more than $12.0 billion. For more information about CDW, you can visit [www.CDW.com](http://www.CDW.com).
Mastering Windows PowerShell Scripting

PowerShell is a network scripting language that provides a set of tools to administer Microsoft products. While PowerShell is based on command-line interactions, it is much more powerful than what the standard command line offers. It has built-in sections of code called cmdlets. They simplify functions that you may need to perform on a system. Using cmdlets greatly reduces the number of lines of code that are required to perform actions, compared to other scripting languages, such as VBScript.

PowerShell is based on the verb-noun naming convention, which allows scripters to declare an action followed by an object to configure. For example, the get-service cmdlet easily designates that you are getting a Windows service. This literal naming convention helps readers quickly learn how to program in PowerShell, as the actions are easily remembered.

Community support for PowerShell has grown astronomically. Not only have large companies adopted PowerShell in their environments, but universities are also regularly teaching PowerShell courses to their students. PowerShell's feature set keeps growing with every release of the product. It is conceivable in the near future that you will be able to fully automate the configuration of every component in a data center. This will remove the needs of multiple engineering specialists to provision networking, storage, firewalls, operating systems builds, and high-availability configurations. It will all be done via PowerShell scripting and the systems will be able to be configured using a singular network language.

This book provides a strong foundation for learning PowerShell using real-world scenarios. You will not only be able to quickly learn how to program in this language, but also be able to produce scripts that you can use in your existing environments. This book will also be a great reference book for you to look back on and revisit as you are coding. It will provide the proper syntax and show you successful ways to implement your code. When you are done with reading this book, you will be well on your way to "mastering PowerShell"!
What This Book Covers

Chapter 1, Variables, Arrays, and Hashes, explores the different data and object containers that you can use in PowerShell. These containers include variables, arrays, and hashes. This chapter provides examples on how to use these containers to store objects.

Chapter 2, Data Parsing and Manipulation, dives into the different data types and how to manipulate them in your scripts. These data type examples include strings, integers, dates, XML, and many more.

Chapter 3, Comparison Operators, evaluates multiple comparison operators and displays how to use each of these comparison operators. This chapter also displays how to leverage implied true and false comparison operators.

Chapter 4, Functions, Switches, and Loops Structures, displays the use of different data structures to perform repeatable actions. It provides examples on how to parse large arrays of data through looping structures and how to include overload parameters in these structures.

Chapter 5, Regular Expressions, explores PowerShell's implementation of regular expressions. It evaluates the built-in comparison operators that provide expression validation and how to create complex expressions.

Chapter 6, Error and Exception Handling and Testing Code, shows you how to create code in a robust manner to avoid exceptions during execution. This chapter explains various built-in error and exception handling techniques, as well as support for legacy systems that don't support PowerShell cmdlet triggers. It also explains the different items to be aware of during the testing cycle of your code.

Chapter 7, Session-based Remote Management, provides an insight into session-based management through PowerShell. It displays how to leverage the built-in WinRM to execute items on remote systems.

Chapter 8, Managing Files, Folders, and Registry Items, displays how to query, create, modify, and delete items in the filesystem and registry. This includes files, folders, registry keys, registry-named values, and properties.

Chapter 9, File, Folder, and Registry Attributes, ACLs, and Properties, dives deep into the interworking of files, folders, and registries. This chapter explains how to set file and folder standards and advanced attributes. It also displays how to manipulate ACLs to set permissions on files, folders, and registry items.

Chapter 10, Windows Management Instrumentation, explains how to use Windows Management Instrumentation (WMI) to query local and remote systems for advanced system information and the different cmdlets that provide access to a system's WMI.
Chapter 11, *XML Manipulation*, explores eXtensible Markup Language (XML) and shows you how to interact with it using PowerShell. This chapter explains the different components that make up a proper XML document and how to interact with these individual components.

Chapter 12, *Managing Microsoft Systems with Powershell*, provides information on how to work with Windows users and groups, Windows services, Windows processes, and the manipulation of Windows features and roles.

Chapter 13, *Automation of the Environment*, explains how to invoke items for use with automation scripts. This chapter explains parent and child relationships because they pertain to linking scripts together. It also explores Desired Configuration Management (DCM) and configuration baselines.

Chapter 14, *Script Creation Best Practices and Conclusion*, provides best practice recommendations for utilizing PowerShell in your environment. This chapter concludes with some final thoughts from the author.
When you are scripting in PowerShell, you will find that a lot of your coding efforts will require the code to be repeated multiple times in the same script. While repeating the same code may help you accomplish the task, there are many other options for coding more efficient scripts. This chapter explores different techniques for which you can reuse code instead of repeating the same code segments within the same script.

In this chapter, you will learn about the following concepts:

- Creation of functions
- Creation of loops
- Creation of switches
- Combining the use of functions, switches, and loops
- Best practices for functions, switches, and loops

**Functions**

When you need to query or execute code more than once, the general rule is that you should create a function to perform the action. Functions are blocks of reusable code, which you can execute multiple times by calling the function's name. You must place a function near the beginning or top of the script. This allows PowerShell to interpret the whole function before you use it later in the code. All other code, including invoking the functions, should follow the functions section. If you call a function that has not yet been parsed by PowerShell, it will throw an exception stating that no such cmdlet or function exists.
Function names can be any word or set of words; however, it is recommended to name the function similar to the verb-noun cmdlet naming syntax. To create a function, you need to use the word `Function` and declare a function name like `display-text`. You then need to enclose the repeatable commands in curly brackets after the function name.

The proper syntax of a function looks like this:

```powershell
Function Display-Text { Write-Host "Showing Text" }
Display-Text
```

The output of this command is shown in the following screenshot:

```
PS C:\> Function Display-Text { Write-Host "Showing Text" }
PS C:\> Display-Text
Showing Text
```

This example displays how to properly declare a function. You first call the `Function` command with the `Display-Text` function name. You then place `Write-Host "Show Text"` in the curly brackets after declaring the function name. You then call the function by typing `Display-Text`. After executing the script, the console will print to the screen the message `Showing Text`.

Functions also allow you to *pass in data* for processing. One of the methods to *pass in data* into a function is to declare variables after the function name in parentheses. This function will then be able to use those variables and the data in those variables within itself. If you want to pass in multiple arguments into a function, you can separate each variable with a comma.

The format to declare a function with parameters in parentheses looks like this:

```powershell
Function Display-Text($variable1,$variable2) {
Write-Host "First Function Argument: $variable1"
Write-Host "Second Function Argument: $variable2"
}
Display-Text "Hello" "Readers"
```
This example displays how to properly declare a function with the parameter in parentheses. You first call the `Function` command with the `Display-Text` function name. You then place the variables, \$variable1 with a comma and \$variable2 in parentheses, before the curly brackets. Inside the curly brackets, you declare `Write-Host "First Function Argument: \$variable1"` and `Write-Host "Second Function Argument: \$variable2"`. You then call the function by typing `Display-Text"Hello" "Readers"`. After executing the script, the console will print to the screen First Function Argument: Hello and Second Function Argument: Readers.

Another method to pass in data to a function is through the use of a parameter block of `Param`. `Param` takes in whatever data you pass into the function and stores that data in declared variables. If you want to pass in multiple arguments into a function, you can separate each variable with a comma. When you are declaring parameters using this method, `Param` needs to be the first item declared after the open curly bracket in a function.

The format to declare a function with `param` looks like this:

```powershell
Function Display-Text { Param($variable1, $variable2)
Write-Host "First Function Argument: \$variable1"
Write-Host "Second Function Argument: \$variable2"
}
Display-Text "Hello" "Readers"
```

The output of this command is shown in the following screenshot:
This example displays how to properly declare a function with parameters in a `Param` block. You first call the `Function` command with the `Display-Text` function name. You then call the `Param` block as the first command inside the curly brackets. Inside the `Param` block, you declare the variables `$variable1` with a comma and `$variable2`. After the `Param` block, you declare `Write-Host "First Function Argument: $variable1"` and `Write-Host "Second Function Argument: $variable2"`. You then call the function by typing `Display-Text` with the arguments `Hello` and `Readers`. After executing the script, the console will print to the screen `First Function Argument: Hello` and `Second Function Argument: Readers`.

The `Param` block is special as it can also accept additional decorators when declaring the variables. The `[Parameter()]` decorator allows you to include additional arguments that enable you to validate variables and even provide help information for variables in functions. When you declare the `Mandatory` argument and set it equal to `$True`, it will require that the variable is used in the function to continue. If you set the `Mandatory` argument to `$False`, it will not be required when using the function. You can also call the `Position` argument, which declares what position the variable will be declared. This means that if you set the `Position` argument to 1, it must be the first argument passed into the function. If you don't use the `Position` argument, you will only be able to pass in the variables using parameter that references the variable name. Another popular argument is the `HelpMessage` argument, which enables you to declare a help message for the individual arguments being passed in. This message is what is displayed in the console when mandatory arguments are missing when a function is being executed. To add multiple parameter arguments in a decorator, you can separate the arguments with commas.

The format to declare a function with `Param` looks with the `[Parameter()]` decorator and parameter arguments looks like this:

```powershell
Function Display-Text {
    #Declare the Parameter Block
    Param(
        #Set The First Parameter as Mandatory with a Help Message
        [Parameter(Mandatory=$True,HelpMessage="Error: Please Enter A Computer Name")]$computername,
        #Set the Second Parameter as Not Mandatory
        [Parameter(Mandatory=$False)]$Message
    )
    Write-Host "First Mandatory Function Argument: $computername"
    Write-Host "Second Function Argument: $Message"
}
```
Display-Text –computername "MyComputerName" "MyMessage"

Display-Text

The output of this command is shown in the following screenshot:

```
$C:\> Function Display-Text {
    #Declare the Parameter Block
    Param(
        #Set The First Parameter as Mandatory with a Help Message
        [Parameter(Mandatory=$True,HelpMessage="Error: Please Enter A Computer Name")]$computername,
        #Set The Second Parameter as Not Mandatory
        [Parameter(Mandatory=$False)]$message
    )
    #Write Host "First Mandatory Function Argument: $computername"
    #Write Host "Second Function Argument: $message"
    )
    Display-Text -computername "MyComputerName" "MyMessage"
    Display-Text
}
First Mandatory Function Argument: MyComputerName
Second Function Argument: MyMessage

cmdlet Display-Text at command pipeline position 1
Supply values for the following parameters:
(Type '?' for Help.)
computername 1
Error: Please Enter A Computer Name
computername: 
```

This example displays how to create a function using `param` with the `[Parameter()]` decorator and parameter arguments. You first call the `Function` command with the `Display-Text` function name. You then call the `Param` block as the first command inside the curly brackets. Inside the `Param` block, you declare the several parameter arguments for the variables. The first argument you call is the `[Parameter]` decorator, and the `Mandatory=$True` attribute. You then include a comma to accept the second attribute of `HelpMessage="Error: Please Enter A Computer Name"`. You then close the parameter decorator and complete the first `Param` item by defining the `$computername` variable.

You include a comma to create a second `Param` item. This `Param` item uses the `[Parameter]` decorator and the `Mandatory=$False` attribute. You close the parameter decorator and complete the second `Param` item by defining the `$message` variable. You then close the `Param` block.

After the `Param` block, you declare `Write-Host "First Mandatory Function Argument: $computername"` and `Write-Host "Second Function Argument: $message"`. You then call the function by typing `Display-Text` with the arguments `-computername "MyComputerName"` and "MyMessage". You also call `Display-Text` without any arguments.
After executing the script, the console will first print to the screen First Mandatory Function Argument: MyComputerName and Second Function Argument: MyMessage. When the script executes the second Display-Text, however, it will print on the screen cmdlet Display-Text at command pipeline position 1 Supply values for the following parameters: Type !? for help. It will then prompt for the computername argument. If you type !? and press Enter, you will see the HelpMessage attribute displayed in the console with the message Error: Please Enter A Computer Name. It will then prompt for the computername argument again until you enter a value.


Functions allow you to pass back data to the section of the script that called the function in the first place. One of the methods with which you can achieve this is with the use of the return command. For example, if after execution of a function you want to pass back the value of $True, you can state return $True. The section of the script that executed the command will then be able to use the value of $True to execute on. You may also use the write-output cmdlet, which acts like the return command and passes back the values to the script. You could also choose the piping method to pass back data. To use the piping method, you take the output from the function and pipe it to a cmdlet or another section of code.

The format to declare functions that return values to the script looks like this:

```powershell
Function Create-WarningMessage {
    $MyError = "This is my Warning Message!"
    $MyError
}
Function Create-Message { Return "My Return message." }
Function Create-Message2 { Write-Output "My Write-Output message." }
Create-WarningMessage | Write-Warning
Create-Message
Create-Message2
```
The output of this command is shown in the following screenshot:

```powershell
PS C:\> Function Create-WarningMessage {
    $MyError = "This is my Warning Message!"
    $MyError
}

Function Create-Message { Return "My Return message." }

Function Create-Message2 { Write-Output "My Write-Output message." }

Create-WarningMessage | Write-Warning
Create-Message
Create-Message2

WARNING: This is my Warning Message
My Return message.
My Write-Output message.
```

This example displays how to declare functions that return values to the script. You first call the `Function` command with the `Create-WarningMessage` function name. Inside the curly brackets, you create a variable named `$MyError` and set it equal to `This is my Warning Message`. You then call the `$MyError` variable and close the function. You create a second function by using the `Function` command with the `Create-Message` function name. Inside the curly brackets, you use the `Return` command with the message `My Return message`. Finally, you create a third function by using the `Function` command with the `Create-Message2` function name. Inside the curly brackets, you use the `write-output` cmdlet with the text `My Write-Output message`.

When you run the script, you first call the `Create-WarningMessage` function and pipe it to `Write-Warning`. When you do this, the output from `Create-WarningMessage` of `This is my Warning Message!` is passed to the `Write-Warning` via the pipeline, and a warning message of `WARNING: This is my Warning Message!` is printed to the console. You then call the `Create-Message` function, which returns from the function and prints to the screen `My Return message`. Finally, you call the `Create-Message2` function, which passes back the `write-output` cmdlet message and prints to the screen `My Write-Output message`.

If you need to exit a function, you can simply use the `return` command, which gracefully exits the function. This avoids having to stop the whole script by using the `EXIT` or `BREAK` commands.
Looping structures

PowerShell provides a variety of looping structures for evaluating and executing objects. Loops are helpful in situations where you need to take in an array of objects and process the individual values in the array. Subsequently, loops are also helpful in situations where you need to wait for a specific value within an array before proceeding in the script.

There are four main looping structures in PowerShell. These looping structures include Do/While, Do/Until, ForEach, and For. The Do/While looping structure is used to execute a task when a value doesn't equal a specific value. The inverse of this looping structure is Do/Until, where it will keep looping the structure until a value equals a specific value. ForEach is a looping structure that allows you to process each individual object in an array or set of objects. The For loop is typically used to execute a task a set number of times.

To create a new Do/While looping structure, you first start by declaring the Do command. You then place the PowerShell commands that you want to repeat in curly brackets. After closing the curly brackets, you declare the While command with a conditional statement. The condition typically leverages a comparison operator and you tell the loop to repeat when the statement equals a specific condition. Once the evaluation of the condition no longer returns True, the loop will stop.

The format of a Do/While looping structure looks like this:

```powershell
$x = 1
$myVar = $False
Do {
    If ($x -ne "4") {
        Write-Host "This Task Has Looped $x Times"
    }
    If ($x -eq "4") {
        $myVar = $True
        Write-Host "Successfully executed the script $x times"
    }
    $x++
} While ($myVar -eq $False)
```
The output of this is shown in the following screenshot:

```
PS C: \> $x = 1
PS C: \> $myVar = $false
PS C: \> Do {
  If ($x -ne "4") {
    Write-Host "This Task Has looped $x times"
  }
  If ($x -eq "4") {
    $myVar = $true
    Write-Host "Successfully executed the script $x times"
  }
  $x++
}
>> While ($myVar -eq $false)
>>
This Task Has looped 1 times
This Task Has looped 2 times
This Task Has looped 3 times
Successfully executed the script 4 times
```

The preceding script displays the proper usage of the Do/While loop structure. The script starts by declaring a variable $x equal to 1. The $x variable designates that it is the first time you are executing the script. You will then declare the $myVar equal to False to allow the script to execute while the variable is False. The Do clause will then execute while $myVar equals False. With each iteration of the loop, the script will evaluate whether the $x variable equals 4. If it doesn't equal 4, it will write to the console This task has looped $x times. It will increment $x by one value designated by the $x++ command and restart from the beginning of the loop. When $x equals 4, the script will set $myVar value to True and write to the console the message Successfully executed the script $x times. The loop will evaluate $myVar and determine that it no longer equals False and exit the loop.

To create a new Do/Until looping structure, you first start by declaring the Do command. You then place the PowerShell commands that you want to repeat in curly brackets. After closing the curly brackets, you declare the Until command with a conditional statement. The condition typically leverages a comparison operator and you tell the loop to repeat until the statement equals a specific condition. Once the evaluation of the condition no longer returns False, the loop will stop.
When you are creating looping structures, it's inevitable that you will accidentally create an infinite looping structure. When you do, you may be flooding your console with text or create a large amount of data. To pause a loop, press *Pause* on your keyboard. If you want to continue, you can hit *Enter* on the keyboard. To completely exit a loop, you can press the key combination of *Ctrl + C* in the console window. This will break the looping structure.

The format of a `Do/Until` looping structure looks like this:

```powershell
$x = 1
$myVar = $False
Do {
    If ($x -ne "4") {
        Write-Host "This Task Has Looped $x Times"
    }
    If ($x -eq "4") {
        $myVar = $True
        Write-Host "Successfully executed the script $x times"
    }
    $x++
} Until ($myVar -eq $True)
```

The output of this is shown in the following screenshot:

```
PS C:\> $x = 1
PS C:\> $myVar = $False
PS C:\> Do {
>     If ($x -ne "4") {
>         Write-Host "This Task Has Looped $x Times"
>     }
>     If ($x -eq "4") {
>         $myVar = $True
>         Write-Host "Successfully executed the script $x times"
>     }
>     $x++
> } Until ($myVar -eq $True)
> This Task Has Looped 1 Times
> This Task Has Looped 2 Times
> This Task Has Looped 3 Times
> Successfully executed the script 4 times
```
The preceding script displays the proper usage of the Do/Until loop structure. The script starts by declaring a variable $x equal to 1. The $x variable designates that it is the first time you are executing the script. You will then declare $myVar equal to False to allow the script to execute while the variable is False. The Do clause will then execute until $myVar equals True. With each iteration of the loop, the script will evaluate whether the $x variable equals 4. If it doesn't equal 4, it will write to the console This task as Looped $x Times. It will increment $x by one value designated by the $x++ command and restart from the beginning of the loop. When $x equals 4, the script will set the $myVar value to True and write to the console Successfully executed the script $x times. The loop will evaluate $myVar and determine that it no longer equals True and exit the loop. You will see that the Do/Until loop structure is declared exactly as the previous script; however, PowerShell interprets the Until statement as a conditional statement to continue until something equals a value.

The.ForEach loop structure has a very simple construct. The ForEach looping structure is declared by calling ForEach. You then specify parentheses containing a variable, the word in, and typically a second variable that contains an array of data. This may look like ($object in $array). While the $array variable typically contains a set of objects, the $object variable is considered the processing variable. This variable enables you to access each object in the array and its properties. After you declare the variables in the parentheses, you place the PowerShell code you want to repeat in curly brackets. In the instance that you want to interact with the individual objects in the array, you can leverage the processing variable in your PowerShell code.

While creating the ForEach loop for processing variables and arrays, it's important to name the variables reflective of what you are processing. If you had a list of account numbers, you could create variables that reflected ForEach ($account in $accountNumbers). This will reduce confusion while reading your scripts.

The format of a ForEach looping structure looks like this:

```powershell
$users = "Mitch", "Ted", "Tom", "Bill"
ForEach ($user in $users) {
    Write-host "Hello $user!"
}
```
The output of this is shown in the following screenshot:

```
PS C:\> $users = "Mitch", "Ted", "Tom", "Bill"
PS C:\> ForEach ($user in $users) {
      Write-host "Hello $user!"
    }
Hello Mitch!
Hello Ted!
Hello Tom!
Hello Bill!
```

In this example, you define an array named $users with the individual values of Mitch, Ted, Tom, and Bill. You then declare the ForEach loop with a processing variable of $user. This loop will then process each $user in the array $users and write to the console the message Hello $user!. The $user variable will be reflecting the current value of the current object that the loop is processing.

The For looping structure has a slightly more complex construct. You first start by declaring the For command. You then declare three required sections of code separated by semicolons and enclose these sections in parentheses. The first section of code is declaring a variable that will interact with the looping structure. This typically is a number that is incremented or decreased as the looping structures proceeds through the loops. This variable must contain a value, otherwise the looping structure will not proceed. This value can be either defined before you enter the looping structure, or when you are declaring the looping structure itself.

The second section of code is the conditional statement, which tells the loop to continue while this statement is True. Once the statement is False, the loop terminates. The last section of code tells the loop to either increment or decrease the first variable and by how many. You can use shorthand to increase a variable by entering $variable++ to increase the value 1, or you can decrease the value by performing a math operation like $variable - 1. After enclosing the required sections of the For loop structure, you enclose the code you want to repeat in curly brackets.

The format of a For looping structure may look like this:

```
For ($x = 1; $x -lt "5"; $x++) {
      write-host "Hello World! Loop Attempt Number: $x"
    }
```
The output is shown in the following screenshot:

```
PS C:\> For ($x = 1; $x -lt "5"; $x++) {
>> write-host "Hello World! Loop Attempt Number: $x"
>> }
Hello World! Loop Attempt Number: 1
Hello World! Loop Attempt Number: 2
Hello World! Loop Attempt Number: 3
Hello World! Loop Attempt Number: 4
```

This example displays how to properly use a `For` looping structure. You first start by declaring the `For` command. You then declare the required sections for the structure; you start by defining `$x` equal to the value 1, which starts the first loop at the value of 1. You then declare the conditional statement of loop while `$x` is less than 5. In the last required section, you declare `$x++`, which increments the `$x` variable by 1 in every loop throughout the structure. You then declare the PowerShell command `Write-host "Hello World! Loop Attempt Number: $x"` in curly brackets. When you run this script, the `For` looping structure will loop 4 times writing to the console the message `Hello World! Loop Attempt Number: $x`, where `$x` equals the iteration of the script's loop.

```
Also, it is important to remember that the Do/While, Do/Until, and For loop structures do not increment the `$x` variable until it processes once. This is why you set the `$x` variable to 1 when you build the construct as it's implied that the first run through the loop has already executed.
```

### Switches

Switches enable you to quickly test multiple scripting scenarios without actually writing `if` statements with comparison operators. Switches are the most efficient flow control commands as they can quickly funnel data into different code sections based on an item. The `Switch` command allows you to evaluate the contents of a single variable and execute subsequent tasks based on the value of the variable. Switches also have a `default` value that is used by the switch when none of the values equal any of the suggested values in the switch statement. To invoke the `Switch` command, you declare `Switch ($variableToEvaluate)`. The second part of the `Switch` command is to declare potential values that the `$variableToEvaluate` could be, as shown here:

```
$x = "that"
Switch ($x) {
```
Functions, Switches, and Loops Structures

this { write-host "Value $x equals this." }
that { write-host "Value $x equals that." }
Default { write-host "Value Doesn't Match Any Other Value" }
}

The output of this is shown in the following screenshot:

PS C:\> $x = "that"
PS C:\> Switch ($x) {
  >> this { write-host "Value $x equals this." }
  >> that { write-host "Value $x equals that." }
  >> Default { write-host "Value Doesn't Match Any Other Value" }
  >> }
  >>
  Value that equals that.

The preceding script displays the proper construct of a Switch command. This example starts by setting the $x variable to that. It then enters the Switch construct that compares the $x variable to the suggested values. In this example, the $x variable equals that, after which the Switch will then write to the console the message Value that equals that. If the value of $x was set to this, it would write to the console the message Value this equals this. Last, if the value of $x is set to anything other than this or that, it would write to the console Value Doesn't Match Any Other Value.

Combining the use of functions, switches, and loops

There may be instances where you will want to combine the use of the different structures explained in this chapter. The example that you will create is a simple menu system that can be modified for use within your scripts. This script will prompt for your interaction and perform actions based on your response, as shown here:

# A Menu System for Use With This Example
Function menu-system {
  Write-host "*********************************************"
  Write-Host "* Please Make A Selection Below:"
  Write-Host "***
  Write-host "***
Write-host "**"
Write-host "**"
Write-host "*****************************************************************************"
Write-host "**
Write-host "Please Make A Selection:
# Prompt for a User Input.
$x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
# A Switch to Evaluate User Input.
Switch ($x.character) {
    1 { write-host "Option 1: User Permissions Backed Up." }
    2 { write-host "Option 2: User Permissions Deleted." }
    3 { write-host "Option 3: User Permissions Restored." }
    Default {
        return $True
    }
}
# A Loop Structure That will Loop Until $Restart doesn't equal true.
Do {
    $restart = Menu-system
    If ($restart -eq $True) {
        cls
        write-host "!! Invalid Selection: Please Try Again"
        write-host ""
    }
} Until ($restart -ne $True)
The output of this is shown in the following screenshot:

```powershell
PS C:\> Function menu-system {
    Write-host "**************************************************************************"
    Write-host "# Please Make A Selection Below:"
    Write-host "#"
    Write-host "# [1] Backup User Permissions."
    Write-host "#"
    Write-host "#"
    Write-host "# [3] Restore User Permissions."
    Write-host "#**************************************************************************"
    Write-host ""
    Write-host "Please Make A Selection:"
    # Prompt for a user input.
    $x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")
    # A switch to evaluate user input.
    Switch ($x.character) {
        1 { write-host "Option 1: User Permissions Backed Up." }
        2 { write-host "Option 2: User Permissions Deleted." }
        3 { write-host "Option 3: User Permissions Restored." }
        Default { return $true }
    }
    # A loop structure that will loop until $restart doesn't equal true.
    Do {
        $restart = menu-system
        if ($restart -eq $true) {
            cls
            write-host "!! Invalid Selection: Please Try Again"
            write-host ""
        }
    } Until ($restart -ne $true)
    #**************************************************************************
    # Please Make A Selection Below:
    
    
    
    
    #**************************************************************************
    Please Make A Selection:
```
This script displays the proper syntax to create a menu system within PowerShell. It first starts by declaring a function named `menu-system`. The `menu-system` function prints to the console instructions on how to use the `menu-system`. It then pauses and waits for user interaction by declaring $x = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown"). When you press a key on the keyboard, the input is set to the $x variable and the script continues. The script then enters the `Switch` command and evaluates the input character ($x.character) against the options that you set up. If you press 1, the console will write Option 1: User Permissions Backed Up and exit the `Switch`. If you press 2, the console will write Option 2: User Permissions Deleted and exit the `Switch`. If you press 3, the console will write Option 3: User Permissions Restored and exit the `Switch`. If you press any other keys than 1, 2, or 3, the script will return $True.

This script also leverages a `Do/Until` loop to restart the `menu-system` method each time the user presses an invalid key. The `Do` loop is entered and will execute the `menu-system` method and catches any returns from the method into the $restart variable. Upon successful key entry from the method, the method will write to the console and exit the method. When the method exits, it doesn't return any data to the $restart variable and the $restart variable will be blank. Since this does not equal $True, the `Do/Until` loop will successfully exit the script. Inversely, if the user doesn't enter a correct value, the Method will return $True and set the $restart variable to $True. The `if` statement will evaluate to be $True, clear the screen using the `cls` command, write to the console !! Invalid Selection: Please Try Again, write to the console a line spacer "", and restart at the top of the `Do/Until` loop structure.

**Best practices for functions, switches, and loops**

When you are scripting, you will find that you frequently need to utilize functions, loops, switches, and methods. Each of these code structures enable you to produce code faster so that you don't have to repeat code within your script. As you work with each of these structures, there are several best practices that you can follow.
Best practices for functions
There are a few recommended steps that can be followed to obtain optimum performance from functions. They are listed as follows:

• If you need to execute a sequence of code more than once, you should create a function. This will allow you to quickly repeat the same action without significantly increasing the size of the script.
• If you need to pass information into a function for processing, you should leverage arguments. Arguments will need to be declared in the order by which the function will use them.
• If you need to pass information back from a function, you should utilize the return command. When used with arguments, it allows you to input data, manipulate data, and return it to a variable for use in other areas of the script.
• Functions need to be declared in the script before you use them. When you are stacking multiple functions in a script, place the functions that will be used first near the top of the file, and the others can follow based on the execution order.
• When you are creating new functions, they should be named as "verb-noun". This will allow for other people to quickly read your scripts and determine what action is being performed. The most common verbs are get-, set-, write-, delete-, read-, new-, replace-, insert-, add-, show-, and remove-.

Best practices for looping structures and switches
As you are working with looping structures and switches, there are several recommended best practices that will ensure scripting success, as shown here:

• It is recommended to keep the looping structures positive in nature. Use Do/While and Do/Until with the -eq conditional operator. This will promote performing actions until a variable equals a value or performing an action while a variable equals a value. Positive conditional operators make reading the script much easier and avoid double negative statements.
• While the For looping structure works well for iterative processing of multiple values, it is recommended to leverage the ForEach looping structure. While both looping structures will achieve the same output, ForEach has a much easier format to read.
• When you are declaring variables for use with the ForEach looping structure, it is one of the best practices to use words as variables. For example, you can declare ForEach($user in $list). This makes it clear that you want to process each $user in the $list. This is much better than stating ForEach($x in $y) from a legibility standpoint.

• When you need to create multiple if statements in your script to evaluate a variable, you should leverage the use of switches.

• When you declare the multiple switch options, it is important to create only the necessary values that require action, and set the default value for all other values. This will reduce the complexity of your switch statements.

Summary
This chapter explored some of the fundamental components that are required for creating PowerShell scripts. These components include functions, loops, and switches. Each of these structures has a purpose within your scripts and can reduce the amount of effort in creating your scripts.

During this chapter, you explored how to create the structure of functions. You also learned how to feed arguments into these scripts and return values from a function.

The chapter also explained how you can create different types of looping structures. These looping structures include Do/Until, Do/While, ForEach, and For. The Do/Until loop was designed to execute until a variable equals a value. The Do/While loop will execute while a variable equals a value. The ForEach loop will execute for each object in an array. The final looping structure is For, which will execute for a set number of times as defined in the initial structure of the loop.

You went through the process of creating a Switch. You learned that switches are used in place of multiple "if" statements to evaluate what the contents of a variable are. You also learned that switches have a default value; if a switch doesn't match any of the criteria, it will execute the default section of code.
After learning about the core fundamentals of these components, we pulled the chapter together with an example on how to leverage functions, looping structures, and switches together for creating a simple menu system. This chapter ends by explaining multiple best practices that can be leveraged for the use of functions, loops, and switches.

The next chapter explores regular expressions. Regular expressions enable you to validate data syntax and structure. Regular expressions are frequently used with comparison operators, functions, loops, and switches to do advanced validation of data. You will learn how to leverage regular expressions within your PowerShell scripts.
Where to buy this book

You can buy Mastering Windows PowerShell Scripting from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.