Comprehensive PowerShell Guide Notes

For Windows VM Environments

DADI REVANTH

May 15, 2025

Contents

1 Getting Started with PowerShell

PowerShell is an essential tool for Windows administration, automation, and development. This section covers the basics of accessing and updating PowerShell.

1.1 Starting PowerShell

To start PowerShell on a Windows computer:

1. Press the Windows key or click the Start button.
2. Type `PowerShell` in the search bar.
3. Click on the "Windows PowerShell" or "PowerShell 7" (or similar) application from the search results.

By default, Windows PowerShell (version 5.1 or older) should be available on modern Windows operating systems. PowerShell 7+ is a separate, cross-platform installation.

1.2 Checking PowerShell Version

To check the currently running PowerShell version, open a PowerShell console and execute the following command:

```
1 $PSVersionTable.PSVersion
```

Listing 1: Check PowerShell Version

The output will display the Major, Minor, Build, and Revision numbers. For example, Windows PowerShell 5.1 might show something like `5.1.22621.xxxx`.

1.3 Installing or Updating PowerShell

While Windows PowerShell 5.1 is built-in, PowerShell 7+ (built on .NET Core/ .NET 5+) is the modern, cross-platform version and receives active updates and new features. It's highly recommended for new development and DevOps practices. As of early 2024, PowerShell 7.4 is a Long-Term Support (LTS) version.

To install a newer version like PowerShell 7.4:

1. Go to the official Microsoft PowerShell GitHub releases page (https://github.com/PowerShell/PowerShell/releases).
2. Find the desired version (e.g., 7.4.x).
3. Download the MSI package appropriate for your operating system architecture (e.g., `PowerShell-7.4.x-win-x64.msi` for 64-bit Windows or `PowerShell-7.4.x-win-x86.msi` for 32-bit Windows).
4. Run the downloaded MSI installer.
5. **Installation Location**: The default installation location is typically `C:Files` (for version 7). It is generally recommended to leave this as is. PowerShell 7 installs side-by-side with Windows PowerShell 5.1, so you can use both.
6. **Installation Options**:
   - *Add PowerShell to Path Environment Variable*: Usually enabled by default and recommended.

- *Register Windows Event Logging Manifest*: Useful for script logging.
- *Enable PowerShell Remoting*: Allows remote execution of PowerShell commands. Often useful, but consider security implications.
- *Add 'Open here' context menus to Explorer*: Adds "Open PowerShell 7 here" to the right-click menu in File Explorer.

It's generally safe to leave the default installation options.

7. After installation, you can launch the new PowerShell version (e.g., "PowerShell 7" from the Start Menu).
8. Confirm the version by running `$PSVersionTable.PSVersion` again in the new PowerShell 7 console. It should reflect the newly installed version.

Extra Info:

- PowerShell 7+ can also be installed via Winget, Chocolatey, or from the Microsoft Store.
- Installing PowerShell 7 does not remove Windows PowerShell 5.1. They can coexist.

2 Uses and Environment

PowerShell is more than just a command line; it's a powerful task automation and configuration management framework from Microsoft, built on the .NET platform.

2.1 Key Usages

- **Automating Administrative Tasks**: Managing user accounts and groups in Active Directory, configuring network settings, managing Windows services and processes, working with files, folders, and registry settings.
- **Scripting**: PowerShell offers a full-featured scripting language to automate complex or repetitive tasks. Scripts are saved as `.ps1` files.
- **Remote Management**: Manage one or hundreds of computers remotely using PowerShell Remoting (based on WinRM). This is crucial for server administration.
- **Cloud Management**: PowerShell is a primary tool for managing cloud resources in Azure (via the Az module), AWS (via AWS Tools for PowerShell), and Google Cloud.
- **CI/CD Pipelines**: Used extensively in Continuous Integration/Continuous Deployment pipelines for build, test, and deployment automation (e.g., in Azure DevOps, Jenkins).
- **Data Manipulation**: PowerShell can easily work with various data formats like XML, JSON, CSV, making it useful for data processing and reporting.
- **Configuration Management**: Through Desired State Configuration (DSC), PowerShell can define and enforce the configuration of systems.

2.2 PowerShell 7 and Integrated Scripting Environment (ISE)

Windows PowerShell (versions up to 5.1) included the PowerShell ISE, a graphical editor for writing and debugging scripts. **PowerShell 7 does not include the ISE.**

Reasons for not including ISE in PowerShell 7:

- **Architectural Differences**: PowerShell 7 is built on .NET Core (now just .NET 5/6/7/8+), which is cross-platform. Windows PowerShell ISE is built on the Windows-specific .NET Framework and has tight coupling with Windows Presentation Foundation (WPF), making it difficult to port or integrate directly.
- **Cross-Platform Compatibility**: PowerShell 7 is designed to run on Windows, macOS, and Linux. The ISE is a Windows-only application.
- **Focus on VS Code**: Microsoft's strategic direction for PowerShell scripting and development is centered around Visual Studio Code (VS Code) with the PowerShell extension.

For coding PowerShell 7 scripts, the recommended tool is **Visual Studio Code (VS Code)** with the official PowerShell extension.

3 Setting up Visual Studio Code for PowerShell

VS Code is a free, lightweight, yet powerful source code editor that runs on your desktop and is available for Windows, macOS, and Linux.

3.1 Installing Visual Studio Code

1. Go to the official VS Code website: https://code.visualstudio.com/.
2. Download the installer for your operating system (e.g., User Installer for Windows 10/11).
3. Run the installer.
4. Accept the license agreement.
5. **Installation Location**: The default location is usually fine.
6. **Select Additional Tasks**:
   - *Create a desktop icon* (optional).
   - *Add "Open with Code" action to Windows Explorer file context menu* (recommended).
   - *Add "Open with Code" action to Windows Explorer directory context menu* (recommended).
   - *Register Code as an editor for supported file types* (optional, but can be useful).
   - *Add to PATH* (requires shell restart, highly recommended).
7. Click "Install" and follow the on-screen instructions.

3.2 Initial Setup in VS Code

1. Launch VS Code. You might be prompted to choose a color theme (the default Dark+ is popular).
2. **Install the PowerShell Extension**: This is crucial for PowerShell development.
   - Click on the Extensions view icon in the Activity Bar on the side of VS Code (looks like square blocks, or use `Ctrl+Shift+X`).
   - Search for "PowerShell" in the search bar.
   - Find the official extension by Microsoft (usually the first result, check publisher is "Microsoft").
   - Click "Install". It should complete in a few seconds. This extension provides rich PowerShell language support, IntelliSense, debugging, and integration with the PowerShell Integrated Console.

3.3 Managing Scripts and Folders

1. **Open a Folder**: It's best practice to work within a project folder.
    - Click on the Explorer view icon (top-most icon in the Activity Bar, looks like files, or use `Ctrl+Shift+E`).
    - Click "Open Folder".
    - You can create a new folder (e.g., `C:`) and then select it.
2. **Trust Authors**: VS Code has a Workspace Trust feature. When you open a folder, you might be asked if you trust the authors of the files in this folder. For your own script folders, it's generally safe to check the box and click "Yes, I trust the authors."
3. **Create a New PowerShell Script File**:
    - In the Explorer view, with your folder open, click the "New File" icon (a page with a plus sign).
    - Name your file with a descriptive name and ensure it has the `.ps1` extension. For example, `MyFirstScript.ps1`.

Extra Info: The PowerShell extension in VS Code provides an "Integrated Console" which is a PowerShell session running within VS Code. This is where your scripts will run and where you can type ad-hoc commands.

4 Basic Scripting and Commands

This section introduces fundamental PowerShell commands and how to run scripts.

4.1 Writing and Running Code

You can write PowerShell code directly into your `.ps1` file in VS Code. A simple command to start with is `Get-Date`, which retrieves the current date and time.

```
# This is a comment
Get-Date # Retrieves the current date and time
Write-Output "Hello, PowerShell World!"
```

Listing 2: Simple PowerShell Command

Running Scripts/Commands in VS Code:

- **Run Script (F5)**: Press F5 or click the "Run and Debug" icon (play button with a bug) and then the green play button (or select "Run Without Debugging"). This typically runs the entire script file currently open and active in the editor.
- **Run Selection (F8)**: Highlight the specific line(s) of code you want to execute, then press F8, or right-click and choose "Run Selection".
- **Using the Integrated Console**: You can also type commands directly into the PowerShell Integrated Console at the bottom of VS Code and press Enter.

The output of your commands or scripts will appear in the Terminal panel (specifically, the PowerShell Integrated Console) at the bottom of the VS Code window.

4.2 Cmdlets (Command-Lets)

In PowerShell, commands are known as **Cmdlets** (pronounced "command-lets"). They are the heart of PowerShell. Cmdlets follow a standardized naming convention: **Verb-Noun**.

- **Verb**: Specifies the action the cmdlet performs (e.g., `Get`, `Set`, `Add`, `Remove`, `New`, `Start`, `Stop`).
- **Noun**: Specifies the entity the cmdlet acts upon (e.g., `Service`, `Process`, `Item` (for files/-folders), `Content`, `Date`, `EventLog`).

Examples:

- `Get-Date`: Gets the current date and time.
- `Get-Service`: Gets the services on the computer.
- `Start-Service`: Starts a service.
- `Get-Process`: Gets the currently running processes.
- `Set-Location`: Changes the current working directory (similar to `cd`).

```
Get-Service -Name "WinRM" # Get information about the WinRM service

Get-Process -Name "notepad" -ErrorAction SilentlyContinue # Get notepad process
    if running
```

Listing 3: Example Cmdlets

If your script has multiple commands, running the entire script will execute them sequentially. If you run a selection, only the highlighted code will execute.

Extra Info:

- Cmdlets often have parameters to modify their behavior. Parameters are usually preceded by a hyphen (e.g., `-Name`, `-Path`).
- PowerShell is generally case-insensitive for cmdlet names and parameters, but it's good practice to follow the PascalCase convention (e.g., `Get-Service`, not `get-service`). Values provided to parameters might be case-sensitive depending on the parameter and underlying system.

5 VS Code Editor Customization

VS Code is highly customizable. Here are a few common tweaks:

5.1 Layout Customization

- **Toggle Sidebar**: The Activity Bar (Explorer, Search, Source Control, Run, Extensions) can be hidden/shown using `Ctrl+B` or `View > Appearance > Show Side Bar`.
- **Toggle Panel (Terminal/Output/Problems)**: The bottom panel can be hidden/shown using `Ctrl+J` or `View > Appearance > Show Panel`.
- You can drag and drop views to different locations in the sidebar or panel.

5.2 Font Size

To change the font size of the text editor:

1. Go to `File > Preferences > Settings` (or `Ctrl+,`).
2. In the search bar at the top of the Settings tab, type "font size".
3. Under `Text Editor > Font`, find the `Editor: Font Size` setting.
4. Adjust the numerical value (e.g., 14, 16) to your preference. Changes are applied immediately.

5.3 Color Theme

To change the editor's color theme:

1. Go to `File > Preferences > Theme > Color Theme` (or `Ctrl+K Ctrl+T`).
2. A dropdown list of installed themes will appear. You can preview them by hovering or using arrow keys.
3. Select a theme (e.g., "Light+", "Solarized Dark", "Monokai").
4. You can also browse for additional themes from the VS Code Marketplace via the Extensions view.

6 Finding Commands and Getting Help

You don't need to memorize all PowerShell commands. PowerShell has excellent built-in help and discovery features.

6.1 Finding Commands with `Get-Command`

- **List all available commands**:

```
Get-Command

```

Listing 4: List all commands

This will output a very long list of cmdlets, functions, and aliases.

- **Find commands related to a specific Noun**: Use the `-Noun` parameter.

```
Get-Command -Noun Service  # Lists cmdlets like Get-Service, New-Service, etc.
Get-Command -Noun Process  # Lists cmdlets like Get-Process, Start-Process, etc.

```

Listing 5: Find commands by Noun

- **Find commands related to a specific Verb**: Use the `-Verb` parameter.

```
Get-Command -Verb Get      # Lists all "Get" cmdlets
Get-Command -Verb Install  # Lists cmdlets like Install-Module, Install-Script

```

Listing 6: Find commands by Verb

- **Using Wildcards**: You can use wildcards (`*`) with `-Noun`, `-Verb`, or directly with `-Name`.

```
Get-Command -Name *Service* # Finds commands with "Service" anywhere in
    their name
Get-Command -Noun *Item*    # Finds commands with Nouns like "Item", "
    ItemProperty"

```

Listing 7: Find commands with wildcards

6.2 Getting Help with `Get-Help`

The `Get-Help` cmdlet provides detailed information about any PowerShell command.

- **Basic Help**:

```
Get-Help Get-Service

```

Listing 8: Basic help for a cmdlet

- **Detailed Help**: Provides more comprehensive information including parameter descriptions and more examples.

```
Get-Help Get-Service -Detailed

```

Listing 9: Detailed help

- **Full Help**: The most comprehensive help, including parameter attributes, input/output types, and notes.

```
Get-Help Get-Service -Full

```

Listing 10: Full help

- **Examples Only**: Shows practical examples of how to use the cmdlet.

```
Get-Help Get-Service -Examples

```

Listing 11: Show examples

- **Show Online Help**: Opens the full help page for the cmdlet in your default web browser, which is often the most up-to-date.

```
Get-Help Get-Service -Online

```

Listing 12: Online help

- **Updating Help Files**: Help content can be updated. Run PowerShell as an Administrator and execute:

```
Update-Help

```

Listing 13: Update help files (run as Administrator)

  Do this periodically to ensure you have the latest help information.

6.3 Aliases

Aliases are shorter, alternative names for cmdlets. They can make interactive use faster but can reduce script readability if overused or if others are not familiar with them.

- Example: `gsv` is an alias for `Get-Service`. `gps` is an alias for `Get-Process`. `cd` is an alias for `Set-Location`. `ls` or `dir` are aliases for `Get-ChildItem`.
- Running an alias executes the full cmdlet.

```
gsv # Same as Get-Service

```

Listing 14: Using an alias

- **List all available aliases**:

```
Get-Alias

```

Listing 15: List all aliases

- **Find alias for a specific cmdlet**:

```
Get-Alias -Definition Get-Service

```

Listing 16: Find alias for a cmdlet

- **Find cmdlet for a specific alias**:

```
Get-Alias -Name gsv

```

Listing 17: Find cmdlet for an alias

Best Practice: While aliases are convenient for interactive use in the console, it's generally recommended to use the full cmdlet names in scripts for better clarity and maintainability, especially when sharing code or working in a team.

7 Variables

Variables are used to store data that can be used and manipulated in your scripts.

7.1 Naming Conventions

While PowerShell is flexible, common naming conventions improve readability:

- **Camel Case**: First word starts with a lowercase letter, subsequent words start with an uppercase letter (e.g., `$myVariable`, `$userName`). Often used for local variables within scripts/functions.
- **Pascal Case**: Every word starts with an uppercase letter (e.g., `$MyVariable`, `$UserName`). Often used for parameters or global/script-scope variables.
- **Snake Case**: Words are separated by underscores (e.g., `$my_variable`). Less common in PowerShell but still valid.

Consistency within a project or team is key.

7.2 Declaration and Assignment

- Variables in PowerShell are declared using the dollar symbol (`$`) followed by the variable name.
- The equals sign (`=`) is used for assignment.

```
# String variable
$greeting = "Hello, PowerShell User!"

# Integer variable
$userCount = 150

# Boolean variable
$isProcessed = $true

# You can also explicitly type variables, though often not necessary
[string]$explicitString = "This is explicitly a string"
[int]$explicitNumber = 42
```

Listing 18: Variable declaration and assignment

7.3 Using and Printing Variables

To use or print the value of a variable, simply use its name (including the `$` symbol).

```
$name = "Alice"
$age = 30

# Print variable values
Write-Output $name
Write-Output $age

# Use variables in strings (string interpolation with double quotes)
Write-Output "User: $name, Age: $age"
# Output: User: Alice, Age: 30

# Using subexpressions for properties or complex expressions in strings
Write-Output "Next year, $name will be $($age + 1)."
# Output: Next year, Alice will be 31.
```

Listing 19: Using variables

7.4 Quotes and Data Types

- **Double Quotes (`" "`)**: Creates an *expandable string*. Variables within double quotes are replaced by their values (variable interpolation). Special characters (like `n` for newline, `t` for tab) are also interpreted.

```
$city = "New York"
$message = "The city is $city." # $city is replaced
Write-Output $message # Output: The city is New York.

```

Listing 20: Double quotes

- **Single Quotes (`' '`)**: Creates a *literal string*. Variables and special characters within single quotes are treated as literal characters and are not interpreted or expanded.

```powershell
$city = "London"
$message = 'The city is $city.' # $city is NOT replaced
Write-Output $message # Output: The city is $city.

```

Listing 21: Single quotes

- **Numbers**: Typing a number without quotes treats it as a numerical data type (e.g., integer, double). PowerShell infers the type. `$number = 06` will be stored as the integer `6`.
- **Automatic Type Detection**: PowerShell is dynamically typed. It usually infers the data type of a variable from the value assigned to it. You can check a variable's type using the `.GetType()` method.

```powershell
$myText = "Sample"
$myNumber = 123.45
$myFlag = $true

Write-Output $myText.GetType().FullName    # System.String
Write-Output $myNumber.GetType().FullName  # System.Double
Write-Output $myFlag.GetType().FullName    # System.Boolean

```

Listing 22: Checking data types

7.5 Properties and Methods

Variables in PowerShell are objects (even simple ones like strings or numbers). Objects have properties (attributes or data associated with the object) and methods (actions the object can perform).

- Access properties and methods using the dot (`.`) symbol after the variable name.
- **Properties**: Represent attributes. Example: `$myString.Length` (gets the number of characters in a string). In VS Code suggestions, properties often have a wrench-like icon.
- **Methods**: Represent actions. They are followed by parentheses `()`. Example: `$myString.ToUpper()` (converts string to uppercase), `$myArray.Contains(value)`. In VS Code suggestions, methods often have a cube-like icon.

```powershell
$myString = "PowerShell"

# Property: Length
Write-Output "Length of '$myString' is: $($myString.Length)" # Output: 10

# Method: ToUpper()
$upperString = $myString.ToUpper()
Write-Output "Uppercase: $upperString" # Output: POWERSHELL

# Method: Replace()
$replacedString = $myString.Replace("Shell", "Ful")
Write-Output "Replaced: $replacedString" # Output: PowerFul

# Method: GetType()
Write-Output "Data type: $($myString.GetType().Name)" # Output: String
```

Listing 23: Using properties and methods

7.6 Clearing the Console

The `Clear-Host` cmdlet (alias `clear` or `cls`) clears the display in the PowerShell console.

```
Clear-Host
# or
cls
```

Listing 24: Clearing the console

Extra Info:

- PowerShell has several built-in "automatic variables" like `$true`, `$false`, `$null`, `$_` (current pipeline object), `$Error` (array of recent errors).
- To see all variables in the current session: `Get-Variable`.
- To remove a variable: `Remove-Variable -Name myVariable` or `Clear-Variable -Name myVariable` (removes value, keeps variable as `$null`).

8 Arithmetic Operators

PowerShell supports standard arithmetic operators for numerical calculations.

- **Addition**: `+`
- **Subtraction**: `-`
- **Multiplication**: `*`
- **Division**: `/`
- **Modulo** (remainder): `%`

```
$num1 = 15
$num2 = 4

$sum = $num1 + $num2
Write-Output "Sum ($num1 + $num2): $sum" # Output: 19

$difference = $num1 - $num2
Write-Output "Difference ($num1 - $num2): $difference" # Output: 11

$product = $num1 * $num2
Write-Output "Product ($num1 * $num2): $product" # Output: 60

$quotient = $num1 / $num2
Write-Output "Quotient ($num1 / $num2): $quotient" # Output: 3.75

$remainder = $num1 % $num2
Write-Output "Remainder ($num1 % $num2): $remainder" # Output: 3

# Order of operations (PEMDAS/BODMAS) is respected
$result = (5 + 3) * 2
Write-Output "(5 + 3) * 2 = $result" # Output: 16
```

Listing 25: Arithmetic operations

Extra Info:

- PowerShell can perform arithmetic with different numeric types, often converting them implicitly.

- The `+` operator can also be used for string concatenation and array concatenation.

```
$string1 = "Power"
$string2 = "Shell"
$combinedString = $string1 + $string2
Write-Output $combinedString # Output: PowerShell

$array1 = 1, 2
$array2 = 3, 4
$combinedArray = $array1 + $array2
Write-Output $combinedArray # Output: 1 2 3 4 (each on a new line usually)

```

Listing 26: Overloaded + operator

9 Boolean Variables

Boolean variables can hold one of two values: true or false. They are fundamental for conditional logic.

- PowerShell has built-in boolean values: `$true` and `$false`. These are case-insensitive in usage (`$True`, `$TRUE` also work) but the canonical forms are `$true` and `$false`.
- The data type of a boolean variable is `System.Boolean`.

```
$isSuccess = $true
$hasFailed = $false

Write-Output "Operation success: $isSuccess"
Write-Output "Operation failure: $hasFailed"
Write-Output "Type of ‘$isSuccess: $($isSuccess.GetType().FullName)" # Output:
    System.Boolean

if ($isSuccess) {
    Write-Output "The condition was true."
}
```

Listing 27: Boolean variables

Important Note: `$true` and `$false` are **read-only automatic variables**. You cannot assign a new value to them.

```
# This will produce an error:
# $true = "some value"
# Error: Cannot overwrite variable true because it is read-only or constant.
```

Listing 28: Attempting to modify $true will error

Extra Info - Truthiness: In PowerShell conditional statements (like `if`), various values can be evaluated as true or false:

- `$null` is false.
- Zero (`0`) is false. Non-zero numbers are true.
- Empty string (`""` or `''`) is false. Non-empty strings are true.
- Empty array (`@()`) is false. Arrays with elements are true.
- Empty hash table (`@{}`) is false. Hash tables with key-value pairs are true.

```
if (0) { Write-Output "0 is true" } else { Write-Output "0 is false" } # Output:
    0 is false
if (1) { Write-Output "1 is true" } else { Write-Output "1 is false" } # Output:
    1 is true
if ("") { Write-Output "Empty string is true" } else { Write-Output "Empty
   string is false" } # Output: Empty string is false
if ("text") { Write-Output "'text' is true" } else { Write-Output "'text' is
   false" } # Output: 'text' is true
```

Listing 29: Truthiness examples

10 Comparison Operators

PowerShell uses hyphen-based operators for comparisons, unlike the symbolic operators (`==`, `!=`, `>`, `<`) found in many other programming languages. These operators return a boolean value (`$true` or `$false`).

- **Equals**: `-eq` (case-insensitive by default for strings)
- **Not Equals**: `-ne` (case-insensitive by default for strings)
- **Greater Than**: `-gt`
- **Greater Than or Equals**: `-ge`
- **Less Than**: `-lt`
- **Less Than or Equals**: `-le`

```
$numA = 10
$numB = 20
$strA = "Hello"
$strB = "hello"

Write-Output "$numA -eq $numB : $($numA -eq $numB)"       # False
Write-Output "$numA -ne $numB : $($numA -ne $numB)"       # True
Write-Output "$numA -gt $numB : $($numA -gt $numB)"       # False
Write-Output "$numA -lt $numB : $($numA -lt $numB)"       # True
Write-Output "$numB -ge $numA : $($numB -ge $numA)"       # True
Write-Output "$numA -le $numA : $($numA -le $numA)"       # True

# String comparison (case-insensitive by default)
Write-Output "'$strA' -eq '$strB' : $($strA -eq $strB)"   # True (default case-
   insensitivity)
Write-Output "'$strA' -ne '$strB' : $($strA -ne $strB)"   # False
```

Listing 30: Comparison operators

Case-Sensitive Comparisons: For case-sensitive string comparisons, prefix the operator with `c`:

- `-ceq` (case-sensitive equals)
- `-cne` (case-sensitive not equals)
- `-clt`, `-cgt`, etc. (for lexicographical comparison)

```
$strA = "Apple"
$strB = "apple"

Write-Output "'$strA' -ceq '$strB' : $($strA -ceq $strB)" # False (case-sensitive)
Write-Output "'$strA' -cne '$strB' : $($strA -cne $strB)" # True (case-sensitive)
```

Listing 31: Case-sensitive comparison

Other Comparison Operators:

- `-like`, `-notlike`: Wildcard comparison (e.g., `"PowerShell" -like "Power*"`). Case-insensitive by default (`-clike`, `-cnotlike` for case-sensitive).
- `-match`, `-notmatch`: Regular expression matching. Case-insensitive by default (`-cmatch`, `-cnotmatch` for case-sensitive). Fills the `$Matches` automatic variable.
- `-contains`, `-notcontains`: Checks if a collection contains a specific value. Exact match.
- `-in`, `-notin`: Checks if a value is present in a collection. Exact match.

```
$myString = "Windows PowerShell Guide"
Write-Output "'$myString' -like '*PowerShell*' : $($myString -like '*PowerShell*')" # True

$myArray = "Red", "Green", "Blue"
Write-Output "'$myArray' -contains 'Green' : $($myArray -contains 'Green')" # True
Write-Output "'Yellow' -in `$myArray : $('Yellow' -in $myArray)" # False
```

Listing 32: Other comparison operators

11 Arrays

An array is a data structure that can hold a collection of multiple items (elements). These items can be of the same data type or mixed data types.

11.1 Creating Arrays

- Assign multiple values, separated by commas, to a variable.
- The `@()` syntax can also be used to define an array, especially useful for an empty array or an array with a single element where ambiguity might arise.

```
# Array of integers
$numbers = 1, 2, 3, 4, 5
Write-Output $numbers # Prints each element on a new line

# Array of strings
$colors = "Red", "Green", "Blue"
Write-Output $colors

# Mixed data type array
$mixedArray = 1, "Hello", $true, (Get-Date)
Write-Output $mixedArray

# Empty array
$emptyArray = @()
Write-Output "Empty array count: $($emptyArray.Count)" # Output: 0
```

```
16
17 # Array with a single element (using @() for clarity)
18 $singleElementArray = @("Apple")
19 Write-Output $singleElementArray
```

Listing 33: Creating arrays

The data type of an array of integers might be `System.Object[]` if it's mixed, or `System.Int32[]` if all elements are integers and PowerShell infers this. You can strongly type arrays: `[int[]]$intOnlyArray = 1,2,3`.

11.2 Array Properties and Accessing Elements

- `.Count` or `.Length` Property: Returns the number of items in the array.

```
1 $fruits = "Apple", "Banana", "Cherry"
2 Write-Output "Number of fruits: $($fruits.Count)" # Output: 3
3
```

Listing 34: Array count

- **Accessing Elements by Index**: Array elements are accessed by their index number. **Indexing starts from 0**.

```
1 $fruits = "Apple", "Banana", "Cherry"
2 Write-Output "First fruit: $($fruits[0])"   # Output: Apple
3 Write-Output "Second fruit: $($fruits[1])"  # Output: Banana
4 Write-Output "Third fruit: $($fruits[2])"   # Output: Cherry
5
```

Listing 35: Accessing array elements by index

The note in the source '"$aaccessesthefirstitem$"'$isincorrect.$'a' would display all elements. '$a[0]$'$accessesthefirstitem.$**Negative Indices** : $Accesselementsfromtheendofthearray.$`-1`$isthelastelem$

`Create an array of numbers` $sequence = 1..5 Createsanarray : 1, 2, 3, 4, 5 Write-Output" Sequence :`sequence"`

11.3 Modifying Arrays

- **Changing an Element**: You can change the value of an element at a specific index.

```
1 $colors = "Red", "Green", "Blue"
2 $colors[1] = "Yellow" # Change "Green" to "Yellow"
3 Write-Output $colors # Output: Red Yellow Blue
4
```

Listing 36: Modifying an array element

- **Adding Elements (+= operator)**: The `+=` operator can be used to add elements to an array. **Important**: PowerShell arrays are technically fixed-size. When you use `+=`, PowerShell actually creates a *new* array that includes the original elements plus the new ones, and then assigns this new array back to the variable. This can be inefficient for very large arrays or frequent additions.

```
1 $myArray = 1, 2
2 $myArray += 3       # Add a single element
3 $myArray += 4, 5   # Add multiple elements
4 Write-Output $myArray # Output: 1 2 3 4 5
5
```

Listing 37: Adding elements to an array

Extra Info - Dynamic Collections: For scenarios requiring frequent additions or removals, traditional PowerShell arrays can be inefficient. It's often better to use more dynamic collection types from .NET:

- `System.Collections.ArrayList`: A dynamic list.

```
$list = New-Object System.Collections.ArrayList
$null = $list.Add("Apple") # .Add() returns the index, $null suppresses it
$null = $list.Add("Banana")
$list.Insert(1, "Orange") # Insert at specific index
$list.Remove("Apple")
Write-Output $list

```

Listing 38: Using ArrayList

- `System.Collections.Generic.List[T]`: A strongly-typed generic list (preferred for type safety).

```
$stringList = New-Object "System.Collections.Generic.List[string]"
$stringList.Add("Cat")
$stringList.Add("Dog")
# $stringList.Add(123) # This would error because it's a list of strings
Write-Output $stringList

```

Listing 39: Using Generic List

12 ForEach Loop (Statement)

The `ForEach` looping construct (statement) is used to iterate through items in a collection, such as an array or the keys/values of a hash table.

12.1 Syntax

The basic syntax is:

```
ForEach ($singularItemVariable in $collectionVariable) {
    # Code block to execute for each item
    # Use $singularItemVariable to refer to the current item
}
```

- `$singularItemVariable`: A temporary variable that holds the current item from the collection during each iteration of the loop. You choose this name (e.g., `$item`, `$color`, `$number`).
- `$collectionVariable`: The array or collection you are looping through.
- The code within the curly braces {} is executed once for each item in the collection.

12.2 Examples

```
$planets = "Mercury", "Venus", "Earth", "Mars"

ForEach ($planet in $planets) {
    Write-Output "Current planet: $planet"
}
# Output:
# Current planet: Mercury
```

```
# Current planet: Venus
# Current planet: Earth
# Current planet: Mars
```

Listing 40: ForEach loop with an array of strings

```
$numbers = 1, 2, 3, 4, 5

ForEach ($num in $numbers) {
    $square = $num * $num
    Write-Output "The square of $num is $square."
}
# Output:
# The square of 1 is 1.
# The square of 2 is 4.
# ...and so on.
```

Listing 41: ForEach loop with calculations

Extra Info - `ForEach-Object` Cmdlet vs. `ForEach` Statement: PowerShell has two distinct `ForEach` constructs:

1. **`ForEach` Statement** (as described above): This is a language keyword for iteration. It's generally faster for simple array iteration.
2. **`ForEach-Object` Cmdlet** (alias `foreach` or `%`): This cmdlet is designed to work in the **pipeline**. It processes objects one by one as they come from the pipeline.

```
$numbers = 1, 2, 3

# Using ForEach-Object in a pipeline
$numbers | ForEach-Object {
    $result = $_ * 10 # $_ represents the current pipeline object
    Write-Output "Pipeline item $_ multiplied by 10 is $result"
}

# Short alias %
$numbers | % { Write-Output "Item: $_" }
```

Listing 42: ForEach-Object cmdlet example

While their names are similar, their use cases and performance characteristics differ. For iterating over an existing array outside of a pipeline, the `ForEach` statement is typically preferred. For processing pipeline input, `ForEach-Object` is used.

13 Hash Tables (Dictionaries)

A **hash table**, also known as a dictionary or associative array in other languages, is a collection that stores data as **key-value pairs**. Each key must be unique and is used to retrieve its associated value.

13.1 Creating Hash Tables

Hash tables are created using the `@` symbol followed by curly braces {}. Key-value pairs are defined inside, with the syntax `Key = Value`, separated by semicolons or newlines.

- Keys are typically strings. If a key contains spaces or special characters, it should be enclosed in quotes (single or double).
- Values can be of any data type (strings, numbers, booleans, arrays, other hash tables, objects).

```powershell
# Simple hash table
$userInfo = @{
    Name = "Alice Wonderland"
    UserID = 1001
    IsAdmin = $false
    Department = "Development"
}

# Hash table with quoted key
$appConfig = @{
    "Application Name" = "My Awesome App"
    Version = "2.1.0"
    Port = 8080
    Features = @("Login", "Dashboard", "Reporting") # Value can be an array
}

Write-Output $userInfo
Write-Output $appConfig
```

Listing 43: Creating hash tables

13.2 Accessing Values

Values are accessed using the key, enclosed in square brackets `[]`.

- Key access is **not case-sensitive** by default.
- You can also use dot notation (`$hashTable.KeyName`) if the key doesn't contain spaces or special characters and doesn't conflict with built-in member names.

```powershell
$userInfo = @{
    Name = "Alice Wonderland"
    UserID = 1001
    IsAdmin = $false
}

Write-Output "User's Name: $($userInfo["Name"])"      # Output: Alice Wonderland
Write-Output "User's ID (case-insensitive key): $($userInfo["userid"])" # Output: 1001
Write-Output "Is Admin (dot notation): $($userInfo.IsAdmin)" # Output: False

# Accessing a non-existent key returns $null
Write-Output "User's Location: $($userInfo["Location"])" # Output: (nothing, $null)
```

Listing 44: Accessing hash table values

You can retrieve multiple values by providing a comma-separated list of keys within the square brackets:

```powershell
$settings = @{ One = 1; Two = 2; Three = 3 }
$subset = $settings["One", "Three"]
Write-Output $subset # Output: 1 3 (each on a new line)
```

Listing 45: Accessing multiple values

13.3 Modifying Hash Tables

- **Changing a Value**: Assign a new value to an existing key.
- **Adding a New Key-Value Pair**: Assign a value to a new key.

- **Removing a Key-Value Pair**: Use the `.Remove()` method.

```
$serverConfig = @{
    ServerName = "SRV01"
    IPAddress = "192.168.1.10"
    Role = "WebServer"
}

# Change a value
$serverConfig["IPAddress"] = "192.168.1.11"
# Or using dot notation
# $serverConfig.IPAddress = "192.168.1.11"
Write-Output "Updated IP: $($serverConfig.IPAddress)"

# Add a new key-value pair
$serverConfig["OS"] = "Windows Server 2022"
# Or
# $serverConfig.Add("Status", "Online") # Using .Add() method

Write-Output $serverConfig

# Remove a key-value pair
$serverConfig.Remove("Role")
Write-Output "After removing Role:"
Write-Output $serverConfig
```

Listing 46: Modifying hash tables

13.4 Iterating Through Hash Tables

You can loop through a hash table using the `ForEach` statement. To access both keys and values, iterate through the `.Keys` property or use the `.GetEnumerator()` method.

```
$capitals = @{
    USA = "Washington D.C."
    UK = "London"
    France = "Paris"
}

# Iterate using .Keys
Write-Output "--- Iterating using .Keys ---"
ForEach ($countryKey in $capitals.Keys) {
    $capitalCity = $capitals[$countryKey]
    Write-Output "The capital of $countryKey is $capitalCity."
}

# Iterate using .GetEnumerator() (provides DictionaryEntry objects)
Write-Output "--- Iterating using .GetEnumerator() ---"
ForEach ($entry in $capitals.GetEnumerator()) {
    Write-Output "Country: $($entry.Key), Capital: $($entry.Value)"
}
```

Listing 47: Iterating through a hash table

The source note ‘ForEach ($item in settings) ... ‘ where ‘$settings$ ‘$is a hash table implies that$ ‘item‘ will be a ‘DictionaryEntry‘ object (or similar structure) from which ‘$item.Key$ ‘and ‘item.Value‘ can be accessed. This is what happens when you call ‘$settings.GetEnumerator()$‘. $If you iterate$ ‘settings‘ directly, the behavior might be less intuitive for beginners. Iterating through ‘$settings.Keys$‘ $is often clearer$.

13.5 Other Useful Properties and Methods

- `.Keys`: Returns a collection of all keys in the hash table.

- `.Values`: Returns a collection of all values in the hash table.
- `.Count`: Returns the number of key-value pairs.
- `.ContainsKey("keyName")`: Checks if a specific key exists. Returns `$true` or `$false`. This check is *not case-sensitive* for the key name by default.
- `.ContainsValue(value)`: Checks if a specific value exists.

```
$employee = @{
    ID = "E721"
    Name = "Bob Smith"
    Salary = 60000
}

Write-Output "Keys: $($employee.Keys)"
Write-Output "Values: $($employee.Values)"
Write-Output "Number of pairs: $($employee.Count)"

Write-Output "Contains key 'Name': $($employee.ContainsKey("Name"))"   # True
Write-Output "Contains key 'name': $($employee.ContainsKey("name"))"   # True (case-insensitive)
Write-Output "Contains key 'Department': $($employee.ContainsKey("Department"))"  # False
```

Listing 48: Hash table properties and methods

Extra Info - Ordered Hash Tables: By default, hash tables in PowerShell (version 3.0 and later) are ordered; they maintain the order in which elements were added. If you need to explicitly ensure this or for compatibility with older versions, you can use the `[ordered]` type accelerator:

```
$orderedSettings = [ordered]@{
    First = 1
    Second = 2
    Third = 3
}
# The order First, Second, Third will be preserved when iterating or printing.
Write-Output $orderedSettings.Keys # Output: First Second Third
```

Listing 49: Ordered hash table

14 Custom Objects

A **custom object** in PowerShell allows you to create structured objects with a defined set of properties and their corresponding values. They are very useful for organizing data, returning structured output from functions, or preparing data for cmdlets like `Export-Csv`.

14.1 Creating Custom Objects

The most common way to create a custom object is by using the `[PSCustomObject]` type accelerator with a hash table-like syntax.

- Unlike hash table keys, property names in a `[PSCustomObject]` definition are typically **not enclosed in double quotes** unless they contain spaces or special characters.
- String values for properties still need to be quoted.

```powershell
$book = [PSCustomObject]@{
    Title  = "The PowerShell Way"
    Author = "Jane Coder"
    Year   = 2023
    ISBN   = "978-0123456789"
}

Write-Output $book
# Output will be a table-like structure by default for custom objects
# Title             Author      Year ISBN
# -----             ------      ---- ----
# The PowerShell Way Jane Coder  2023 978-0123456789
```

Listing 50: Creating a custom object

14.2 Accessing Properties

Properties of a custom object are accessed using the **dot (`.`) symbol**, similar to accessing properties of .NET objects. This is a key difference from hash tables, where values are typically accessed using square brackets and keys (e.g., `$hashTable["Key"]`).

```powershell
$person = [PSCustomObject]@{
    FirstName = "John"
    LastName  = "Doe"
    Age       = 35
    Occupation = "Engineer"
}

Write-Output "First Name: $($person.FirstName)"  # Output: John
Write-Output "Age: $($person.Age)"                # Output: 35

# Using properties in a string (requires subexpression $(...))
$details = "User: $($person.FirstName) $($person.LastName), Occupation: $(
    $person.Occupation)"
Write-Output $details
# Output: User: John Doe, Occupation: Engineer
```

Listing 51: Accessing custom object properties

14.3 Difference from Hash Tables

Feature	**Hash Table (`@{}`)**	**Custom Object (`[PSCustomObject]`)**
Creation	`$ht = @{Key = "Val"}`	`$obj = [PSCustomObject]@{Prop = "Val`
Value Access	`$ht["Key"]` or `$ht.Key`	`$obj.Prop` (dot notation primary)
Intended Use	Key-value lookups, configuration	Structured data, object representation
Output Format	Displays as key-value pairs	Displays as a table by default
Mutability	Keys/values can be added/removed/modified after creation.	Properties fixed after creation (though valu To add/remove properties, a new object is

Table 1: Hash Table vs. Custom Object Key Differences

Extra Info:

- Custom objects are instances of `System.Management.Automation.PSCustomObject`.
- You can add methods to custom objects using `Add-Member` if needed, though this is less common for simple data structures.
- Custom objects integrate well with PowerShell's pipeline and cmdlets like `Select-Object`, `Where-Object`, `Export-Csv`, `ConvertTo-Html`, etc.

```powershell
$user = [PSCustomObject]@{
    UserName  = "jdoe"
    FullName  = "John Doe"
    Email     = "john.doe@example.com"
    LastLogin = (Get-Date).AddDays(-1)
}

# Select specific properties
$user | Select-Object UserName, Email
```

Listing 52: Custom object with Select-Object

15 List of Custom Objects (Arrays of Custom Objects)

You can create a collection, specifically an array, where each element is a custom object. This is extremely useful for representing lists of structured data, like a list of users, servers, or products.

15.1 Creating a List of Custom Objects

This is done by creating an array (`@()`) where each element is a `[PSCustomObject]` definition. Elements are separated by commas.

```powershell
$employees = @(
    [PSCustomObject]@{
        EmployeeID = "E001"
        Name       = "Alice Smith"
        Department = "HR"
        Salary     = 60000
    },
    [PSCustomObject]@{
        EmployeeID = "E002"
        Name       = "Bob Johnson"
        Department = "IT"
        Salary     = 75000
    },
    [PSCustomObject]@{
        EmployeeID = "E003"
        Name       = "Carol White"
        Department = "Finance"
        Salary     = 68000
    }
)

# Displaying the list will typically show a table
Write-Output $employees
# Output:
# EmployeeID Name         Department Salary
# ---------- ----         ---------- ------
# E001       Alice Smith  HR          60000
# E002       Bob Johnson  IT          75000
# E003       Carol White  Finance     68000
```

Listing 53: Creating a list of custom objects

The note "When defining multiple custom objects in a list on a single line, they are separated by a semicolon (;)" might refer to a sequence of statements creating objects which are then collected. For defining an array literal as above, commas are the standard separators for array elements. Example of potentially what was meant by semicolon (less common for direct list creation):

```powershell
$obj1 = [PSCustomObject]@{Name="A"}; $obj2 = [PSCustomObject]@{Name="B"}
```

```
$list = $obj1, $obj2 # Here comma is used for array
Write-Output $list
```

Listing 54: Alternative way to build a list (less direct)

The primary method shown with `@(... , ...)` is the standard for initializing an array of custom objects.

15.2 Iterating Through a List of Custom Objects

Use the `ForEach` statement to loop through the list. Inside the loop, the temporary variable will hold the current custom object, and you can access its properties using dot notation.

```
$servers = @(
    [PSCustomObject]@{ Name = "SRVWEB01"; Role = "Web Server"; Status = "Online"},
    [PSCustomObject]@{ Name = "SRVDB01";  Role = "Database Server"; Status = "Offline"},
    [PSCustomObject]@{ Name = "SRVAPP01"; Role = "Application Server"; Status = "Online"}
)

ForEach ($server in $servers) {
    Write-Output "Server Name: $($server.Name)"
    Write-Output "Role: $($server.Role)"
    Write-Output "Status: $($server.Status)"
    if ($server.Status -eq "Offline") {
        Write-Warning "Warning: Server $($server.Name) is $($server.Status)!"
    }
    Write-Output "---" # Separator
}
```

Listing 55: Iterating through a list of custom objects

15.3 Working with Lists of Custom Objects

Lists of custom objects are powerful because they can be easily manipulated by PowerShell's pipeline cmdlets:

```
$products = @(
    [PSCustomObject]@{ Name = "Laptop";  Category = "Electronics"; Price = 1200},
    [PSCustomObject]@{ Name = "Mouse";   Category = "Electronics"; Price = 25 },
    [PSCustomObject]@{ Name = "Keyboard";Category = "Electronics"; Price = 75 },
    [PSCustomObject]@{ Name = "Desk";    Category = "Furniture";   Price = 300 }
)

# Filter for electronics products
$electronics = $products | Where-Object {$_.Category -eq "Electronics"}
Write-Output "Electronic Products:"
Write-Output $electronics

# Sort products by price (descending)
$sortedProducts = $products | Sort-Object -Property Price -Descending
Write-Output "Products Sorted by Price (High to Low):"
Write-Output $sortedProducts

# Select specific properties and create a new list of objects
$productNamesAndPrices = $products | Select-Object Name, Price
Write-Output "Product Names and Prices:"
Write-Output $productNamesAndPrices
```

Listing 56: Filtering and sorting a list of custom objects

This structure is ideal for generating reports (e.g., `Export-Csv`, `ConvertTo-Html`).

16 Pipeline

The **pipeline** is one of PowerShell's most powerful and distinctive features. It allows you to pass the output of one command (cmdlet or function) as the input to another command, enabling complex operations to be built from simpler components.

16.1 How it Works

- The pipeline is represented by the pipe symbol (`|`).
- The basic structure is: `Command1 | Command2 | Command3 ...`
- PowerShell cmdlets are designed to work with objects. When a command outputs objects, these objects are passed one by one through the pipeline to the next command.
- The next command in the pipeline processes these input objects.

16.2 The `$_` (or `$PSItem`) Automatic Variable

Within a script block used by cmdlets that process pipeline input (like `ForEach-Object`, `Where-Object`), the automatic variable `$_` (or its more descriptive alias `$PSItem` in PowerShell 3.0+) represents the **current object** being passed through the pipeline from the previous command.

- `$_`: Traditional, concise.
- `$PSItem`: More verbose, potentially clearer in complex scripts. Both refer to the same thing.

16.3 Examples

1. **String Manipulation**:

```
"hello world from powershell" | ForEach-Object { $_.ToUpper() }
# Output: HELLO WORLD FROM POWERSHELL
# Explanation:
# 1. The string "hello world from powershell" is sent to the pipeline.
# 2. ForEach-Object receives this string. Inside its script block {...}, $_
    represents "hello world from powershell".
# 3. $_.ToUpper() calls the ToUpper() method on the string.

```

Listing 57: Pipeline for string manipulation

2. **Filtering Processes with `Where-Object`** (alias `where`): `Where-Object` filters objects based on a condition.

```
# Get all processes, then filter for those named 'notepad'
Get-Process | Where-Object {$_.Name -eq 'notepad'}

# Using $PSItem (equivalent)
# Get-Process | Where-Object {$PSItem.Name -eq 'notepad'}
# Explanation:
# 1. Get-Process outputs a collection of process objects.
# 2. Each process object is passed to Where-Object.
# 3. For each object, $_ (or $PSItem) refers to the current process object.
# 4. $_.Name -eq 'notepad' checks if the Name property of the current
    process object is 'notepad'.
# 5. If true, Where-Object passes that process object further down the
    pipeline (or to output).

```

Listing 58: Filtering processes

3. **Selecting Specific Properties with `Select-Object`** (alias `select`): `Select-Object` creates new objects with only the specified properties.

```
# Get all processes, then select only their ID and Name properties
Get-Process | Select-Object -Property ID, Name

# You can also select the first N objects or last N objects
# Get-Process | Select-Object -First 5 # Get the first 5 processes

```

Listing 59: Selecting object properties

4. **Working with Files and Folders**: `Get-ChildItem` (aliases `gci`, `ls`, `dir`) gets items in a location.

```
# Get all files (not directories) in C:\Windows (use a safe path for
    testing)
# then filter for files larger than 1MB (1MB = 1024KB = 1024*1024 bytes)
# Be cautious with paths like C:\Windows, it can list many items. Use a
    smaller test directory.
# Get-ChildItem -Path "C:\Path\To\Your\TestFolder" -File | Where-Object {$_
    .Length -gt 1MB} | Select-Object Name, Length
# For example, find large text files in your documents:
# Get-ChildItem -Path $env:USERPROFILE\Documents -Recurse -Filter "*.txt" -
    File | Where-Object {$_.Length -gt (500*1024)} | Select-Object FullName
    , Length

```

Listing 60: Filtering files by size

Extra Info - Benefits of the Pipeline:

- **Efficiency**: Objects are often processed one at a time ("streaming"), which can be memory efficient, especially with large datasets, as the entire collection doesn't need to be stored in memory by intermediate steps.
- **Modularity**: Complex tasks can be broken down into a series of simpler, reusable commands.
- **Readability**: Well-structured pipelines can be very clear in their intent.
- **Object-Oriented**: Unlike text-based pipelines in traditional shells (like Bash), PowerShell passes rich .NET objects, preserving data types and structure. This means you don't need to parse text output as much.

Cmdlets that are designed to accept pipeline input typically have parameters that are declared to accept `ValueFromPipeline` or `ValueFromPipelineByPropertyName`. You can see this in the full help for a cmdlet (`Get-Help <CmdletName> -Full`).

17 Conditional Statements

Conditional statements allow your scripts to make decisions and execute different blocks of code based on whether certain conditions are true or false.

17.1 If-ElseIf-Else Statement

This is the most common conditional structure. **Syntax**:

```
if (condition1) {
    # Code block to execute if condition1 is true
}
elseif (condition2) {
    # Code block to execute if condition1 is false AND condition2 is true
}
else {
    # Code block to execute if all preceding conditions (condition1, condition2, etc.) are f
}
```

- The `elseif` and `else` blocks are optional.
- You can have multiple `elseif` blocks.
- Conditions are boolean expressions (evaluate to `$true` or `$false`), often using comparison operators.

```
$temperature = 25 # Celsius

if ($temperature -gt 30) {
    Write-Output "It's hot!"
}
elseif ($temperature -gt 20) { # (20 < temperature <= 30)
    Write-Output "It's warm and pleasant."
}
elseif ($temperature -gt 10) { # (10 < temperature <= 20)
    Write-Output "It's cool."
}
else { # (temperature <= 10)
    Write-Output "It's cold!"
}

$userName = "Admin"
if ($userName -eq "Admin") {
    Write-Output "Welcome, Administrator!"
}
else {
    Write-Output "Welcome, User $userName!"
}
```

Listing 61: If-ElseIf-Else example

Logical Operators: You can combine conditions using logical operators:

- `-and`: Logical AND (both conditions must be true).
- `-or`: Logical OR (at least one condition must be true).
- `-xor`: Logical Exclusive OR (one condition is true, but not both).
- `-not` (or `!`) : Logical NOT (reverses the boolean value).

```
$age = 25
$hasLicense = $true

if (($age -ge 18) -and ($hasLicense -eq $true)) {
    Write-Output "Allowed to drive."
} else {
    Write-Output "Not allowed to drive."
}

```

```powershell
$isWeekend = $false
$isHoliday = $true
if ($isWeekend -or $isHoliday) {
    Write-Output "It's a day off!"
}
```

Listing 62: If statement with logical operators

17.2 Switch Statement

The `Switch` statement provides an alternative to long `If-ElseIf-Else` chains, especially when you are checking a single variable or expression against multiple possible values. **Syntax**:

```
switch ($variableOrExpression) {
    value1 {
        # Code block if $variableOrExpression equals value1
    }
    value2 {
        # Code block if $variableOrExpression equals value2
    }
    value3 # Can also match multiple values
    value4 {
        # Code block if $variableOrExpression equals value3 OR value4
    }
    Default {
        # Code block if none of the above values match
    }
}
```

- The `Default` block is optional and executes if no other conditions match.
- By default, `Switch` performs exact, case-insensitive comparisons for strings.
- After a match, PowerShell typically exits the `Switch` statement. If you want it to continue checking other conditions (fall-through behavior, rare in PowerShell), you would use the `-Continue` parameter on the specific case or the `break` keyword to explicitly exit. (Standard behavior is to execute the block and exit switch).

```powershell
$dayOfWeek = (Get-Date).DayOfWeek.ToString() # Gets "Monday", "Tuesday", etc.

switch ($dayOfWeek) {
    "Monday" {
        Write-Output "Start of the work week."
    }
    "Tuesday"
    "Wednesday"
    "Thursday" {
        Write-Output "It's a weekday."
    }
    "Friday" {
        Write-Output "TGIF! Almost the weekend."
    }
    "Saturday"
    "Sunday" {
        Write-Output "Enjoy the weekend!"
    }
    Default {
        Write-Output "Invalid day specified." # Should not happen with Get-Date
```

```
    }
}
```

Listing 63: Switch statement example

Advanced Switch Features: `Switch` can do more than exact value matching:

- `-Wildcard`: Allows wildcard characters (`*`, `?`) in case conditions.
- `-Regex`: Allows regular expressions in case conditions.
- `-Exact`: Forces case-sensitive exact matching.
- `-File <filepath>`: Reads input from a file, processing each line.
- **Script Blocks as Conditions**: Conditions can be script blocks {`$_ -operator value`}. `$_` refers to the input value.

```
$inputValue = "TestFile123.log"

switch -Wildcard ($inputValue) {
    "*.log" { Write-Output "It's a log file." }
    "Test*" { Write-Output "It starts with Test." } # This might be hit first if "*.log" not specific enough
    Default { Write-Output "Unknown file type." }
}

$number = 75
switch ($number) {
    { $_ -lt 50 } { Write-Output "Number is less than 50." }
    { $_ -ge 50 -and $_ -le 100 } { Write-Output "Number is between 50 and 100 (inclusive)." }
    Default { Write-Output "Number is greater than 100." }
}
```

Listing 64: Switch with -Wildcard and script block condition

18 Loops

Loops are used to execute a block of code repeatedly. PowerShell offers several looping constructs.

18.1 ForEach Loop Statement

Already covered in detail in Section 12. It iterates over items in a collection.

```
$colors = @("Red", "Green", "Blue")
ForEach ($color in $colors) {
    Write-Output "Current color: $color"
}
```

Listing 65: Recap: ForEach loop statement

18.2 For Loop

The `For` loop is a classic loop that repeats a block of code as long as a condition is true. It typically involves an initializer, a condition, and an iterator. **Syntax**:

```
for (initializer; condition; iterator) {
    # Code block to execute
}
```

- `initializer`: Executed once before the loop starts (e.g., `$i = 0`).
- `condition`: Evaluated before each iteration. If true, the loop continues. If false, the loop terminates (e.g., `$i -lt 5`).
- `iterator`: Executed at the end of each iteration (e.g., `$i++`).

```
for ($i = 1; $i -le 5; $i++) {
    Write-Output "Current For loop iteration: $i"
}
# Output:
# Current For loop iteration: 1
# ...
# Current For loop iteration: 5

# Loop backwards
for ($j = 3; $j -ge 1; $j--) {
    Write-Output "Countdown: $j"
}
# Output:
# Countdown: 3
# Countdown: 2
# Countdown: 1
```

Listing 66: For loop example

18.3 While Loop

The `While` loop executes a block of code repeatedly *as long as* a specified condition is true. The condition is checked *before* each iteration. **Syntax**:

```
while (condition) {
    # Code block to execute
    # Ensure the condition eventually becomes false to avoid an infinite loop!
}
```

```
$count = 0
while ($count -lt 3) {
    Write-Output "While loop count: $count"
    $count++ # Increment count to eventually exit the loop
}
# Output:
# While loop count: 0
# While loop count: 1
# While loop count: 2
```

Listing 67: While loop example

18.4 Do-While Loop

The `Do-While` loop is similar to the `While` loop, but the condition is checked *after* the code block has executed. This means the code block will **always run at least once**, even if the condition is initially false. **Syntax**:

```
do {
    # Code block to execute
} while (condition)
```

```
$counter = 5
do {
    Write-Output "Do-While counter: $counter"
    $counter++
} while ($counter -lt 3) # Condition is initially false (5 is not < 3)
# Output:
# Do-While counter: 5 (executes once because condition is checked after)

$anotherCounter = 0
do {
    Write-Output "Do-While anotherCounter: $anotherCounter"
    $anotherCounter++
} while ($anotherCounter -lt 3)
# Output:
# Do-While anotherCounter: 0
# Do-While anotherCounter: 1
# Do-While anotherCounter: 2
```

Listing 68: Do-While loop example

18.5 Do-Until Loop

The `Do-Until` loop is similar to `Do-While`, but it continues to execute the code block *until* the condition becomes true. The code block always runs at least once. **Syntax**:

```
do {
    # Code block to execute
} until (condition)
```

```
$value = 0
do {
    Write-Output "Do-Until value: $value"
    $value++
} until ($value -eq 3) # Loop until value is 3
# Output:
# Do-Until value: 0
# Do-Until value: 1
# Do-Until value: 2
```

Listing 69: Do-Until loop example

18.6 Loop Control: `break` and `continue`

- `break`: Immediately exits the current loop (`For`, `ForEach`, `While`, `Do-While/Until`, `Switch`).
- `continue`: Skips the rest of the current iteration and proceeds to the next iteration of the loop.

```
Write-Output "--- Example with break ---"
for ($i = 1; $i -le 10; $i++) {
    if ($i -eq 5) {
        Write-Output "Breaking loop at $i"
        break # Exit the loop
    }
    Write-Output "Iteration $i"
}
# Output: Iteration 1, 2, 3, 4, Breaking loop at 5

Write-Output "--- Example with continue ---"
for ($j = 1; $j -le 5; $j++) {
```

```
    if ($j -eq 3) {
        Write-Output "Skipping iteration $j with continue"
        continue # Skip to next iteration
    }
    Write-Output "Processing $j"
}
# Output: Processing 1, Processing 2, Skipping iteration 3 with continue,
    Processing 4, Processing 5
```

Listing 70: break and continue in a loop

19 Error Handling (Try-Catch-Finally)

By default, PowerShell scripts often continue executing subsequent lines of code even if an error (a non-terminating error) occurs. The `Try-Catch-Finally` construct provides a structured way to handle potential errors, especially terminating errors or when you make non-terminating errors behave as terminating.

19.1 Error Types in PowerShell

- **Non-Terminating Errors**: These are the default for many cmdlets. An error message is displayed, but the script continues execution. Example: `Get-ChildItem -Path "C:"`.
- **Terminating Errors**: These errors halt the execution of the current command pipeline or script block. Examples include syntax errors, dividing by zero, or when a cmdlet explicitly generates a terminating error.

The `Try-Catch-Finally` block primarily deals with *terminating errors*.

19.2 Syntax

```
try {
    # Code that might cause an error (the "guarded" region)
    # To make a non-terminating error act as terminating within this block,
    # use -ErrorAction Stop on the specific command.
}
catch [OptionalExceptionType1] {
    # Code to run if an error of OptionalExceptionType1 (or any error if type omitted)
    # occurs in the try block.
    # The error object is available as $_ or $PSItem.
}
catch [OptionalExceptionType2] {
    # Another catch block for a different specific exception type.
}
# ... more specific catch blocks ...
catch {
    # A general catch block for any other errors not caught by specific types above.
}
finally {
    # Code that always runs, regardless of whether an error occurred or was caught.
    # Used for cleanup operations (e.g., closing files, releasing resources).
}
```

- The `try` block is required.

- At least one `catch` block or one `finally` block must be present.
- `catch` blocks can be specific to .NET exception types (e.g., `[System.IO.FileNotFoundException]`, `[System.Net.WebException]`). If no type is specified, it catches any terminating error.
- The `finally` block is optional.

19.3 Using -ErrorAction Stop

To make a cmdlet that normally produces a non-terminating error trigger a `catch` block, you must change its error action to `Stop` for that specific command.

```
try {
    Write-Output "Attempting to get content from a non-existent file."
    # Get-Content normally produces a non-terminating error for a missing file.
    # -ErrorAction Stop makes it a terminating error for this command.
    $content = Get-Content -Path "C:\Path\To\NonExistentFile123.txt" -
    ErrorAction Stop
    Write-Output "File content retrieved (this won't be reached if file is
    missing):"
    Write-Output $content
}
catch {
    Write-Warning "An error occurred in the try block!"
    Write-Warning "Error Type: $($_.GetType().FullName)"
    Write-Warning "Error Message: $($_.Exception.Message)"
    # $_ or $PSItem in the catch block is the error record object.
}
finally {
    Write-Output "This is the finally block. It always executes."
}
```

Listing 71: Using -ErrorAction Stop

19.4 Example: Handling File Operations

```
$filePath = "C:\temp\mydata.txt"          # A file that might exist or not
$backupPath = "C:\temp\mydata_backup.txt"

# Create a dummy file for testing success path
# New-Item -Path $filePath -ItemType File -Value "Original content" -Force -
    ErrorAction SilentlyContinue

try {
    Write-Host "Attempting to read file: $filePath"
    $fileContent = Get-Content -Path $filePath -ErrorAction Stop
    Write-Host "File content read successfully."

    Write-Host "Attempting to copy to backup: $backupPath"
    Copy-Item -Path $filePath -Destination $backupPath -Force -ErrorAction Stop
    Write-Host "File backed up successfully to $backupPath"
}
catch [System.IO.FileNotFoundException] {
    Write-Warning "File not found: $($_.Exception.FileName)"
    Write-Warning "Message: $($_.Exception.Message)"
    # Specific action for file not found
}
catch [System.UnauthorizedAccessException] {
    Write-Warning "Access denied for operation."
    Write-Warning "Message: $($_.Exception.Message)"
    # Specific action for access denied
}
```

```
catch { # General catch block for any other errors
    Write-Error "An unexpected error occurred:"
    Write-Error "Exception Type: $($_.Exception.GetType().FullName)"
    Write-Error "Message: $($_.Exception.Message)"
    # You can access the full error record via $_
    # $_ | Format-List * -Force # For detailed error info
}
finally {
    Write-Host "Cleanup phase: File operations completed or attempted."
    # Example: Remove a temporary lock file, close a connection, etc.
    # if (Test-Path "C:\temp\lockfile.tmp") { Remove-Item "C:\temp\lockfile.tmp" }
}
```

Listing 72: Try-Catch-Finally for file operations

Extra Info - `$ErrorActionPreference`: You can set the `$ErrorActionPreference` variable to `"Stop"` at the beginning of your script or function to make ALL non-terminating errors behave as terminating errors within that scope.

```
$OldErrorAction = $ErrorActionPreference
$ErrorActionPreference = "Stop" # Makes all non-terminating errors terminating

try {
    Get-Service -Name "NonExistentService" # This will now throw a terminating error
    Write-Host "Service found." # This line won't be reached
}
catch {
    Write-Warning "Caught an error: $($_.Exception.Message)"
}
finally {
    $ErrorActionPreference = $OldErrorAction # Restore original preference
    Write-Host "ErrorActionPreference restored to $OldErrorAction."
}
```

Listing 73: Using $ErrorActionPreference

Using `-ErrorAction Stop` on individual commands is often preferred for more granular control, but `$ErrorActionPreference = "Stop"` can be useful for certain script-wide behaviors. Be sure to restore its original value in a `finally` block if you change it globally.

20 Real-Time DevOps Practices with PowerShell on Windows VMs

PowerShell is a cornerstone for DevOps practices in Windows environments, especially for managing Windows Virtual Machines (VMs). It enables automation, configuration management, deployment, monitoring, and more.

20.1 Configuration Management with PowerShell DSC

PowerShell Desired State Configuration (DSC) is a management platform in PowerShell that enables you to manage your IT and development infrastructure with configuration as code.

- **Declarative**: You define the desired state of your system (e.g., "IIS should be installed," "this registry key must exist with this value"), and DSC makes it so.
- **Idempotent**: Applying a DSC configuration multiple times yields the same result without unintended side effects.

- **Resources**: DSC uses resources (e.g., `WindowsFeature`, `File`, `Registry`, `Service`) to model a system's components.

Conceptual DSC Configuration Example:

```
# MyDSCConfig.ps1
Configuration WebServerConfig {
    # Import DSC resources if needed (e.g., from PSGallery)
    # Import-DscResource -ModuleName PSDesiredStateConfiguration
    # Import-DscResource -ModuleName xWebAdministration # (example for advanced IIS config)

    Node "localhost" { # Can target remote nodes too
        WindowsFeature IIS {
            Ensure = "Present" # Or "Absent"
            Name   = "Web-Server" # The feature name for IIS
            # IncludeAllSubFeature = $true # Optional
        }

        Registry AppVersion {
            Ensure    = "Present"
            Key       = "HKLM:\SOFTWARE\MyApplication"
            ValueName = "Version"
            ValueData = "1.2.3"
            ValueType = "String"
        }
    }
}

# To compile and apply:
# 1. Run the script to load the configuration: .\MyDSCConfig.ps1
# 2. Compile the configuration into a MOF file: WebServerConfig -OutputPath C:\DSC_Configs
# 3. Apply the configuration: Start-DscConfiguration -Path C:\DSC_Configs -Wait -Verbose -Force
```

Listing 74: DSC Configuration to ensure IIS is installed

DSC can be used in push mode (applying configurations directly) or pull mode (nodes pull configurations from a central DSC Pull Server or Azure Automation DSC).

20.2 Automated Deployments

PowerShell is widely used to automate application deployments to Windows VMs. **Example: Simple Web App Deployment Snippet**:

```
param (
    [string]$WebAppName = "MyWebApp",
    [string]$AppPoolName = "MyWebAppPool",
    [string]$SourcePath = "\\buildserver\artifacts\MyWebApp\latest",
    [string]$DestinationPath = "C:\inetpub\wwwroot\MyWebApp"
)

Import-Module WebAdministration # For IIS cmdlets

Write-Host "Starting deployment of $WebAppName..."

# 0. Ensure destination exists
if (-not (Test-Path $DestinationPath)) {
    New-Item -Path $DestinationPath -ItemType Directory -Force
}

# 1. Stop Application Pool and Website (if they exist)
Write-Host "Stopping IIS components..."
```

```powershell
if (Get-Website -Name $WebAppName -ErrorAction SilentlyContinue) {
    Stop-Website -Name $WebAppName
}
if (Get-WebAppPool -Name $AppPoolName -ErrorAction SilentlyContinue) {
    Stop-WebAppPool -Name $AppPoolName
}

# 2. Backup existing application (optional but recommended)
$backupPath = "C:\Backups\MyWebApp_$(Get-Date -Format 'yyyyMMddHHmmss')"
if (Test-Path $DestinationPath) {
    Write-Host "Backing up current version to $backupPath..."
    Copy-Item -Path $DestinationPath -Destination $backupPath -Recurse -Force -ErrorAction SilentlyContinue
}

# 3. Clear destination folder (or use Robocopy for smarter sync)
Write-Host "Clearing destination folder $DestinationPath..."
Get-ChildItem -Path $DestinationPath | Remove-Item -Recurse -Force

# 4. Copy new application files
Write-Host "Copying new files from $SourcePath..."
Copy-Item -Path "$SourcePath\*" -Destination $DestinationPath -Recurse -Force

# 5. Configure IIS (ensure AppPool and Website exist and are configured)
# This part can be extensive depending on needs
if (-not (Get-WebAppPool -Name $AppPoolName -ErrorAction SilentlyContinue)) {
    New-WebAppPool -Name $AppPoolName -ManagedRuntimeVersion "v4.0" # Or "No Managed Code"
}
if (-not (Get-Website -Name $WebAppName -ErrorAction SilentlyContinue)) {
    New-Website -Name $WebAppName -Port 80 -HostHeader "mywebapp.example.com" `
                -PhysicalPath $DestinationPath -ApplicationPool $AppPoolName
}
# Set-ItemProperty -Path "IIS:\Sites\$WebAppName" -Name physicalPath -Value $DestinationPath
# Set-ItemProperty -Path "IIS:\Sites\$WebAppName" -Name applicationPool -Value $AppPoolName


# 6. Start Application Pool and Website
Write-Host "Starting IIS components..."
Start-WebAppPool -Name $AppPoolName
Start-Website -Name $WebAppName

Write-Host "Deployment of $WebAppName completed successfully."
```

Listing 75: Conceptual web app deployment script

This script would typically be part of a CI/CD pipeline (e.g., triggered by Jenkins, Azure DevOps Pipelines, GitHub Actions).

20.3 Managing Windows Services, Processes, and Scheduled Tasks

Automating the management of services, processes, and scheduled tasks is a common DevOps requirement.

```powershell
# Ensure a critical service is running
$serviceName = "MyApplicationService"
$service = Get-Service -Name $serviceName -ErrorAction SilentlyContinue
if ($service -and $service.Status -ne "Running") {
    Write-Warning "Service $serviceName is $($service.Status). Attempting to start..."
    Start-Service -Name $serviceName
    Start-Sleep -Seconds 5 # Give it a moment to start
```

```
    if ((Get-Service -Name $serviceName).Status -eq "Running") {
        Write-Host "Service $serviceName started successfully."
    } else {
        Write-Error "Failed to start service $serviceName."
    }
} elseif (!$service) {
    Write-Error "Service $serviceName not found!"
} else {
    Write-Host "Service $serviceName is already running."
}

# Create or update a scheduled task for daily log cleanup
$taskName = "DailyLogCleanup"
$scriptToRun = "C:\Scripts\Maintenance\CleanupLogs.ps1"
$action = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-
    NoProfile -ExecutionPolicy Bypass -File `"$scriptToRun`""
$trigger = New-ScheduledTaskTrigger -Daily -At "2:00 AM"
$principal = New-ScheduledTaskPrincipal -UserID "NT AUTHORITY\SYSTEM" -RunLevel
    Highest
$settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -
    DontStopIfGoingOnBatteries -ExecutionTimeLimit (New-TimeSpan -Hours 1)

# Unregister if exists, then register
Get-ScheduledTask -TaskName $taskName -ErrorAction SilentlyContinue | Unregister
    -ScheduledTask -Confirm:$false
Register-ScheduledTask -TaskName $taskName -Action $action -Trigger $trigger -
    Principal $principal -Settings $settings -Description "Cleans up application
     logs daily." -Force
Write-Host "Scheduled task '$taskName' configured."
```

Listing 76: Managing services and scheduled tasks

20.4 Log Management and Monitoring

PowerShell can query Windows Event Logs, parse text-based log files, and integrate with monitoring systems.

```
# Get last 10 errors from Application log
Write-Host "Recent Application Errors:"
Get-WinEvent -LogName Application -MaxEvents 10 | Where-Object {$_.
    LevelDisplayName -eq "Error"} | Format-Table TimeCreated, ProviderName,
    Message -AutoSize -Wrap

# Basic CPU and Memory check
$cpuThreshold = 80 # Percent
$memoryThresholdMB = 500 # Available MB

$cpuUsage = (Get-Counter '\Processor(_Total)\% Processor Time').CounterSamples
    [0].CookedValue
$availableMemory = (Get-Counter '\Memory\Available MBytes').CounterSamples[0].
    CookedValue

Write-Host "Current CPU Usage: $($cpuUsage.ToString('F2'))%"
Write-Host "Available Memory: $($availableMemory.ToString('F0')) MB"

if ($cpuUsage -gt $cpuThreshold) {
    Write-Warning "High CPU Usage detected: $($cpuUsage.ToString('F2'))%"
    # Add alerting logic here (e.g., send email, call webhook)
}
if ($availableMemory -lt $memoryThresholdMB) {
    Write-Warning "Low Available Memory detected: $($availableMemory.ToString('
    F0')) MB"
    # Add alerting logic here
```

```
}
```

Listing 77: Querying event logs and basic monitoring

For more advanced log management, tools like NXLog, Fluentd, or Azure Monitor Agent are often used, but PowerShell can help in their configuration or in ad-hoc analysis.

20.5 Interacting with Cloud Providers (e.g., Azure)

PowerShell modules for cloud platforms are essential for IaC (Infrastructure as Code) and VM management. **Example: Using Azure PowerShell (Az module)**:

```
# Ensure Az module is installed: Install-Module Az -Scope CurrentUser -Force
# Connect to Azure: Connect-AzAccount (interactive login)
# Or use a service principal for automation:
# $credential = Get-Credential # For testing, or use certs/MSI for unattended
# Connect-AzAccount -ServicePrincipal -Tenant $TenantID -ApplicationId $AppID -Credential $credential

$resourceGroupName = "MyProductionRG"
$vmName = "MyWindowsVM01"

# Get VM status
Write-Host "Fetching status for VM '$vmName' in RG '$resourceGroupName'..."
$vm = Get-AzVM -ResourceGroupName $resourceGroupName -Name $vmName -Status -ErrorAction SilentlyContinue
if ($vm) {
    $powerState = ($vm.Statuses | Where-Object {$_.Code -like "PowerState/*"})[0].DisplayStatus
    Write-Host "VM '$($vm.Name)' Power State: $powerState"
    Write-Host "VM Size: $($vm.HardwareProfile.VmSize)"

    # Start VM if stopped
    if ($powerState -eq "VM deallocated" -or $powerState -eq "VM stopped") {
        Write-Host "Starting VM '$($vm.Name)'..."
        Start-AzVM -ResourceGroupName $resourceGroupName -Name $vm.Name
    }
} else {
    Write-Error "VM '$vmName' not found in RG '$resourceGroupName'."
}

# List all VMs in a resource group
# Get-AzVM -ResourceGroupName $resourceGroupName | Select-Object Name, VmId, Location, ProvisioningState
```

Listing 78: Basic Azure VM operations with PowerShell

20.6 Security and Patching

PowerShell can automate security hardening tasks and patch management.

- **Windows Update Management**: Using modules like `PSWindowsUpdate` (third-party, widely used) or built-in COM objects for updates.
- **Security Baseline Checks**: Scripts to verify security configurations (e.g., firewall rules, user rights, audit policies) against a baseline.

```
# PSWindowsUpdate module needs to be installed first from PSGallery
# Install-Module PSWindowsUpdate -Force -SkipPublisherCheck -Scope CurrentUser
Import-Module PSWindowsUpdate -ErrorAction SilentlyContinue

if (Get-Module -Name PSWindowsUpdate) {
```

```
    Write-Host "Checking for Windows Updates..."
    $updates = Get-WindowsUpdate -MicrosoftUpdate # Check against Microsoft Update

    if ($updates.Count -gt 0) {
        Write-Warning "Pending updates found:"
        $updates | Format-Table Title, KB, Size -AutoSize

        # Optionally install:
        # Write-Host "Installing updates..."
        # Get-WindowsUpdate -Install -AcceptAll -AutoReboot -Verbose
        # (Use -AutoReboot with extreme caution in production environments)
    } else {
        Write-Host "No pending updates found."
    }
} else {
    Write-Warning "PSWindowsUpdate module not found. Cannot check for updates via this method."
}
```

Listing 79: Conceptual: Check for pending updates using PSWindowsUpdate

20.7 Remote Management with PowerShell Remoting

PowerShell Remoting (WinRM-based) allows you to run commands and scripts on remote Windows VMs securely.

- **Enable PSRemoting**: On target VMs, run `Enable-PSRemoting -Force` (as Administrator). Ensure WinRM service is running and firewall allows WinRM traffic (TCP 5985 for HTTP, 5986 for HTTPS).
- `Invoke-Command`: Executes commands on one or more remote computers.
- `Enter-PSSession`: Starts an interactive session with a remote computer.

```
$targetVMs = "VM01.domain.local", "192.168.1.105" # Can be hostnames or IPs
$credential = Get-Credential # Prompt for credentials for remote access

Write-Host "Running Get-Service WinRM on remote VMs..."
try {
    Invoke-Command -ComputerName $targetVMs -Credential $credential -ScriptBlock {
        # This script block runs on each remote machine
        param($localMachineName) # Example of passing local variable to remote

        Write-Host "Executing on $($env:COMPUTERNAME) (called from $localMachineName)"
        Get-Service -Name "WinRM" | Select-Object MachineName, Name, Status, StartType
        Get-Uptime # Custom function Get-Uptime needs to exist on remote machine or be defined in scriptblock
    } -ArgumentList $env:COMPUTERNAME -ErrorAction Stop # ArgumentList passes params to scriptblock
} catch {
    Write-Error "Failed to connect or execute command on one or more VMs: $($_.Exception.Message)"
}

# Start an interactive session
# Enter-PSSession -ComputerName "VM01.domain.local" -Credential $credential
# [VM01.domain.local]: PS C:\Users\YourUser\Documents> hostname
# ...
```

```
# [VM01.domain.local]: PS C:\Users\YourUser\Documents> Exit-PSSession
```

Listing 80: Using Invoke-Command for remote execution

Security Considerations for Remoting:

- Use HTTPS for WinRM if possible (requires certificates).
- Limit which users/groups can connect via WinRM (configure session configurations).
- Use Just Enough Administration (JEA) to restrict remote users to only necessary commands.

These examples provide a glimpse into how PowerShell facilitates DevOps on Windows VMs. Effective DevOps relies on combining these PowerShell capabilities with version control (Git), CI/CD tools, monitoring solutions, and a culture of automation and collaboration.