CST8202 – Windows Desktop Support

Lab 4 – PowerShell Command Line Tools

Sequential Learning Manual

Table of Contents

Lab Overview

Purpose

To develop proficiency with PowerShell cmdlets, parameters, arguments, and command pipelines while learning essential system administration tasks including disk management, file operations, and network configuration retrieval.

Learning Journey

This lab follows a logical progression:

1. **Foundation** → Understanding PowerShell basics and setup
2. **Discovery** → Learning to find and use cmdlets (aliases)
3. **Storage** → Hardware management and disk operations
4. **Files** → Directory and file management
5. **Information** → System and network data retrieval
6. **Integration** → Combining concepts in complex operations

Equipment/Resources

- College Approved BYOD Laptop
- Access to Brightspace
- Windows 11 Virtual Machine
- VMware Workstation
- Administrative privileges on the VM

Time Estimate

- **Setup**: 30 minutes
- **Core Exercises**: 2.5-3 hours
- **Bonus Exercise**: 45 minutes
- **Documentation**: 30 minutes
- **Total**: 4-4.5 hours

PowerShell Fundamentals

What is PowerShell?

PowerShell is a command-line shell and scripting language built on the .NET Framework. Unlike traditional command prompts that work with text, PowerShell works with .NET objects, making it incredibly powerful for system administration.

Core PowerShell Concepts

1. Cmdlets (Command-lets)

- **Format**: Verb-Noun syntax (e.g., `Get-Process`, `New-Item`)
- **Always hyphenated** and use approved PowerShell verbs
- **Common verbs**: `Get-` (retrieve), `Set-` (modify), `New-` (create), `Remove-` (delete)

2. Parameters

- **Purpose**: Modify how a cmdlet behaves
- **Format**: Always start with a hyphen (`-`)
- **Examples**: `-Path`, `-Name`, `-Recurse`
- **Types**: Positional (order matters) or named (order doesn't matter)

3. Arguments

- **Purpose**: The values you provide to parameters
- **Types**: Strings, numbers, boolean values, objects
- **Examples**: `"C:\Windows"`, `5`, `$true`

4. Pipeline (`|`)

- **Function**: Passes output from one cmdlet as input to another
- **Key difference**: Objects flow through pipeline, not just text
- **Example**: `Get-Process | Where-Object {$_.CPU -gt 100}`

5. Objects vs. Text

- **Traditional shells**: Work with text strings
- **PowerShell**: Works with .NET objects that have properties and methods
- **Access properties**: Using dot notation `$object.PropertyName`

Command Structure Examples

```powershell
# Basic syntax: Verb-Noun -Parameter Argument
Get-ChildItem -Path "C:\Windows"

# Multiple parameters:
New-Item -ItemType Directory -Path "C:\Temp" -Name "NewFolder"

# Using pipeline:
Get-Process | Where-Object {$_.Name -eq "notepad"} | Stop-Process
```

Pre-Lab Setup

Step 1: Opening PowerShell with Administrative Privileges

Why Administrative Privileges? Many system administration tasks (disk management, network configuration, service management) require elevated permissions. Running PowerShell as Administrator grants these permissions.

Method 1: Windows Key Menu (Recommended)

1. Press `Win + X` (Windows Key + X)
2. Select "Windows PowerShell (Admin)" or "Terminal (Admin)"
3. Click "Yes" when User Account Control (UAC) prompts

Method 2: Run Dialog

1. Press `Win + R` (opens Run dialog)
2. Type `powershell`
3. Press `Ctrl + Shift + Enter` (runs as admin)
4. Click "Yes" when UAC prompts

Method 3: Start Menu

1. Right-click Start button
2. Select "Windows PowerShell (Admin)"
3. Click "Yes" when UAC prompts

Step 2: Verifying Administrative Access

Visual Indicators:

- Window title shows "Administrator"
- Prompt shows system directory: `PS C:\Windows\system32>`
- UAC prompted for permission

Step 3: Understanding PowerShell Interface

Prompt Elements

```
PS C:\Windows\system32>
```

- `PS` → Indicates PowerShell (vs CMD)
- `C:\Windows\system32` → Current working directory
- `>` → Command prompt indicator

Important Built-in Variables

```powershell
$env:USERPROFILE     # Current user's profile (C:\Users\Username)
$env:COMPUTERNAME    # Computer name
$env:USERNAME        # Current username
$PSVersionTable      # PowerShell version information
$pwd                 # Present working directory
```

Step 4: Testing Your Setup

```powershell
# Verify administrative access
whoami /priv

# Check PowerShell version
$PSVersionTable.PSVersion

# Verify current location
Get-Location

# Test basic cmdlet
Get-Date
```

Sequential Exercises

Phase 1: PowerShell Discovery and Aliases

Exercise 1: Discovering Existing Aliases

Learning Focus: Understanding PowerShell's built-in shortcuts and help system

Background: What are Aliases?

Aliases are alternative names for cmdlets that are shorter and easier to type. They help users transition from other shells and speed up common operations.

Common PowerShell Aliases:

- `ls` → `Get-ChildItem` (Unix/Linux familiarity)
- `dir` → `Get-ChildItem` (DOS/CMD familiarity)
- `cd` → `Set-Location`
- `copy` → `Copy-Item`
- `del` → `Remove-Item`

The Get-Alias Cmdlet Deep Dive

```powershell
Get-Alias [-Name] <string[]> [-Exclude <string[]>] [-Scope <string>]
Get-Alias [-Definition] <string[]> [-Exclude <string[]>] [-Scope <string>]
```

Parameter Explanations:

- `-Name`: Find alias by its short name (e.g., "ls")
- `-Definition`: Find aliases by the full cmdlet name (e.g., "Get-ChildItem")
- `-Exclude`: Exclude specific aliases from results
- `-Scope`: Search in specific scope (Global, Local, Script)

Task 1: Find Copy-Item Aliases

Objective: Discover all existing shortcuts for the Copy-Item cmdlet

Required Command Structure:

- 1 cmdlet: `Get-Alias`
- 1 parameter: `-Definition`
- 1 argument: `Copy-Item`

Your Command:

```powershell
Get-Alias -Definition Copy-Item
```

Expected Output:

```
CommandType     Name                Version    Source
-----------     ----                -------    ------
Alias           copy -> Copy-Item
Alias           cp -> Copy-Item
Alias           cpi -> Copy-Item
```

Understanding the Output:

- **CommandType**: Type of command (Alias, Function, Cmdlet)
- **Name**: Short alias → Full cmdlet it represents
- **Version**: Version info (usually empty for aliases)
- **Source**: Where the alias is defined

📸 **Screenshot Required**: Command execution and output showing at least 3 aliases

Exploration Commands:

```powershell
# See all aliases
Get-Alias

# Find what 'ls' does
Get-Alias -Name ls

# Find all aliases containing 'item'
Get-Alias | Where-Object {$_.Definition -like "*Item*"}
```

Exercise 2: Creating Custom Aliases

Learning Focus: Alias creation, management, and scope understanding

The New-Alias Cmdlet

```powershell
New-Alias [-Name] <string> [-Value] <string> [-Description <string>]
    [-Option <ScopedItemOptions>] [-PassThru] [-Scope <string>] [-Force]
```

Parameter Deep Dive:

- `-Name`: Your custom alias name
- `-Value`: The full cmdlet it should execute
- `-Description`: Optional documentation
- `-Option`: Behavior controls (ReadOnly, Constant, etc.)
- `-Scope`: Where to create (Global, Local, Script, Private)
- `-Force`: Overwrite existing aliases

Understanding Alias Scopes

- **Global**: Available everywhere in PowerShell session
- **Local**: Available in current scope and child scopes
- **Script**: Available only within current script
- **Private**: Available only in current scope

Task 2: Create "Dupe" Alias

Objective: Create a custom alias "Dupe" for Copy-Item

Why "Dupe"? It's short for "Duplicate," describing what Copy-Item does.

Your Command:

```powershell
New-Alias -Name Dupe -Value Copy-Item
```

Alternative Syntax (positional parameters):

```powershell
New-Alias Dupe Copy-Item
```

📝 **Document**: The exact command you used

Verification:

```powershell
Get-Alias Dupe
```

Expected Output:

```
CommandType     Name                Version    Source
-----------     ----                -------    ------
Alias           Dupe -> Copy-Item
```

Important Notes:

- Aliases are session-specific (lost when PowerShell closes)
- For permanent aliases, add to PowerShell profile
- Use meaningful, memorable names

Exercise 3: Verifying Alias Creation

Learning Focus: Confirmation techniques and alias management

Task 3: Confirm New Alias Exists

Objective: Verify that your custom alias was successfully added

Your Command:

```powershell
Get-Alias -Definition Copy-Item
```

Expected Output (4 aliases now):

```
CommandType     Name                Version    Source
-----------     ----                -------    ------
Alias           copy -> Copy-Item
Alias           cp -> Copy-Item
Alias           cpi -> Copy-Item
Alias           Dupe -> Copy-Item
```

📸 **Screenshot Required**: Complete output showing all 4 aliases including your "Dupe"

Alias Management Commands:

```powershell
# Remove an alias
Remove-Item -Path Alias:Dupe

# Check if alias exists
Test-Path Alias:Dupe

# Export aliases to file
Export-Alias C:\Temp\MyAliases.csv
```

Phase 2: Hardware Management and Storage

Exercise 4: Hardware Discovery - Finding New Storage

Learning Focus: Hardware identification and Windows storage concepts

Background: Windows Disk Management Concepts

Disk States:

- **Online**: Accessible and usable
- **Offline**: Not accessible (common for new disks)
- **Missing**: Previously configured but no longer detected
- **Failed**: Disk failure, unusable

Partition Styles:

- **MBR (Master Boot Record)**: Traditional (≤2TB, 4 primary partitions)
- **GPT (GUID Partition Table)**: Modern (>2TB, unlimited partitions)
- **RAW**: Uninitialized (new disks)

Health Status:

- **Healthy**: Working normally
- **Warning**: Issues but functional
- **Unhealthy**: Problems may cause data loss

Pre-Task: Adding New Disk to VM

Why? Simulates adding storage to a server - common IT task.

VMware Steps:

1. **Shut down VM properly:**

```powershell
Stop-Computer -Force
```

2. **Add disk in VMware:**
   - Right-click VM → Settings
   - Click "Add..." → Hard Disk → Next
   - Select "SCSI" (recommended) → Next
   - Choose "Create a new virtual disk" → Next
   - Set size: **2 GB**
   - Select "Store as single file" → Finish
3. **Power on VM**

The Get-Disk Cmdlet

```powershell
Get-Disk [[-Number] <UInt32[]>] [-CimSession <CimSession[]>]
```

Key Properties:

- **Number**: Unique disk ID (0, 1, 2...)
- **FriendlyName**: Manufacturer/model info
- **Size**: Total capacity
- **PartitionStyle**: MBR, GPT, or RAW
- **HealthStatus**: Physical condition
- **OperationalStatus**: Current availability

Task 4: Identify New Disk

Objective: Find the disk number of your newly added 2GB disk

Your Command:

```powershell
Get-Disk
```

What to Look For:

- **RAW** partition style (uninitialized)
- **2 GB** size
- Usually gets next available number (Disk 1, 2, etc.)

Expected Output:

```
Number Friendly Name      HealthStatus   OperationalStatus  Total Size Partition Style
------ -------------      ------------   -----------------  ---------- ---------------
0      VMware Virtual...  Healthy        Online             20 GB MBR
1      VMware Virtual...  Healthy        Online              2 GB RAW
```

📝 **Document**: Command used and disk number of your 2GB disk

Advanced Filtering:

```powershell
# Show only RAW disks
Get-Disk | Where-Object {$_.PartitionStyle -eq "RAW"}

# Show only 2GB disks
Get-Disk | Where-Object {$_.Size -eq 2GB}

# Detailed view
Get-Disk | Format-Table Number, FriendlyName, Size, PartitionStyle -AutoSize
```

Exercise 5: Advanced Disk Formatting

Learning Focus: Storage initialization, file systems, and complex parameter usage

Storage Management Process

1. **Initialize**: Prepare disk with partition table
2. **Partition**: Divide disk into logical sections
3. **Format**: Apply file system to partition
4. **Assign Drive Letter**: Make accessible in Windows

File System Comparison

Feature	NTFS	FAT32	exFAT
Max File Size	16 TB	4 GB	16 EB
Security	Yes	No	No
Compression	Yes	No	No
Journaling	Yes	No	No
Windows Default	Yes	No	No
Cross-Platform	Limited	Excellent	Good

Why NTFS?

- Default Windows file system
- Supports large files and drives
- Security permissions
- Built-in compression and encryption
- Journaling prevents corruption

The New-Volume Cmdlet

```powershell
New-Volume [-DiskNumber] <UInt32> [-Size <UInt64>] [-UseMaximumSize]
        [-DriveLetter <Char>] [-FileSystem <String>] [-FriendlyName <String>]
```

Critical Parameters:

- **-DiskNumber**: Which disk to use (from Exercise 4)
- **-FileSystem**: How to organize data (NTFS recommended)
- **-FriendlyName**: Human-readable label (Volume Label)
- **-DriveLetter**: Drive access letter (A-Z, avoid A, B, existing letters)

Task 5: Format New Disk

Objective: Format disk with specific requirements

Specifications:

- **File System**: NTFS
- **Volume Label**: "PSDisk"
- **Drive Letter**: P
- **Disk**: Your number from Exercise 4

Required Structure:

- 1 cmdlet
- 4 parameters
- 4 arguments

Your Command (replace X with your disk number):

```powershell
New-Volume -DiskNumber X -FileSystem NTFS -FriendlyName "PSDisk" -DriveLetter P
```

What Happens Behind the Scenes:

1. Disk initialization (RAW → GPT/MBR)
2. Single partition creation (full disk)
3. NTFS formatting
4. Drive letter assignment
5. Volume label setting

Expected Output:

```
DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining    Size
----------- ------------ -------------- --------- ------------ ----------------- -------------    ----
P           PSDisk       NTFS           Fixed     Healthy      OK                1.99 GB          2 GB
```

📝 **Document**: Your exact command with correct disk number

Verification:

```powershell
# Test drive accessibility
Test-Path P:\

# Detailed volume info
Get-Volume -DriveLetter P

# Open in File Explorer
explorer P:\
```

Exercise 6: Storage Verification with Filtering

Learning Focus: Result filtering and verification techniques

Object Filtering Concepts

PowerShell's `Where-Object` enables sophisticated filtering:

- **Comparison operators**: `-eq`, `-ne`, `-gt`, `-lt`, `-like`, `-match`
- **Logical operators**: `-and`, `-or`, `-not`
- **Collection operators**: `-contains`, `-in`

The Get-Volume Cmdlet

Shows all volumes (drive letters) with their properties:

- **DriveLetter**: Assigned letter
- **FileSystemType**: NTFS, FAT32, etc.
- **DriveType**: Fixed, Removable, CD-ROM
- **HealthStatus**: Volume condition

Task 6: Verify Volume Creation

Objective: Confirm volume exists while excluding CD-ROM drives

Your Command:

```powershell
Get-Volume | Where-Object {$_.DriveType -ne "CD-ROM"}
```

Understanding the Filter:

- `Get-Volume` → Gets all volumes
- `|` → Pipes to filter
- `Where-Object {...}` → Filters based on condition
- `$_.DriveType -ne "CD-ROM"` → Excludes CD-ROM drives
- `$_` → Refers to current object in pipeline

Expected Output:

```
DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining    Size
----------- ------------ -------------- --------- ------------ ----------------- -------------    ----
C                   NTFS          Fixed     Healthy      OK                15.2 GB   20 GB
P          PSDisk      NTFS          Fixed     Healthy      OK                 1.99 GB    2 GB
```

📝 **Document**: Command used to verify volume creation excluding CD-ROMs

Alternative Filtering Methods:

```powershell
# Filter by file system
Get-Volume | Where-Object {$_.FileSystemType -eq "NTFS"}

# Filter by drive letter pattern
Get-Volume | Where-Object {$_.DriveLetter -match "[A-Z]"}

# Multiple conditions
Get-Volume | Where-Object {$_.DriveType -eq "Fixed" -and $_.FileSystemType -eq "NTFS"}
```

Phase 3: File System Operations

Exercise 7: Directory Management

Learning Focus: File system navigation and directory creation

Directory Creation in PowerShell

The `New-Item` cmdlet creates various item types:

- **Directory**: Folders
- **File**: Empty files
- **SymbolicLink**: Symbolic links
- **HardLink**: Hard links

Understanding Path Types

- **Absolute Path**: Full path from root (C:\Users\Name\Documents)
- **Relative Path**: Path from current location (.\Documents)
- **Environment Variables**: System-defined paths ($env:USERPROFILE)

Path Construction Methods

```powershell
# Method 1: String concatenation
"$env:USERPROFILE\Documents\Lab4"

# Method 2: Join-Path (preferred - handles separators)
Join-Path $env:USERPROFILE "Documents\Lab4"

# Method 3: .NET method
[System.IO.Path]::Combine($env:USERPROFILE, "Documents", "Lab4")
```

Task 7: Create Lab4 Directory

Objective: Create "Lab4" directory in your Documents folder

Recommended Approach (absolute path):

```powershell
New-Item -ItemType Directory -Path "$env:USERPROFILE\Documents\Lab4"
```

Alternative Approaches:

```powershell
# Navigate first, then create
Set-Location "$env:USERPROFILE\Documents"
New-Item -ItemType Directory -Name "Lab4"

# Using Join-Path
New-Item -ItemType Directory -Path (Join-Path $env:USERPROFILE "Documents\Lab4")
```

Expected Output:

```
    Directory: C:\Users\[USERNAME]\Documents

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        [DATE]   [TIME]                Lab4
```

📝 **Document**: Exact command you used

Verification:

```powershell
# Check if directory exists
Test-Path "$env:USERPROFILE\Documents\Lab4"

# List contents of Documents
Get-ChildItem "$env:USERPROFILE\Documents"
```

Exercise 8: System Information and Output Redirection

Learning Focus: System information retrieval and file output techniques

PowerShell Version Information

`$PSVersionTable` contains comprehensive version details:

- **PSVersion**: PowerShell version number
- **PSEdition**: Desktop (5.1) or Core (6.0+)
- **GitCommitId**: Source control information
- **OS**: Operating system details
- **Platform**: Hardware platform

Output Redirection Deep Dive

Redirection Operators:

```powershell
>     # Redirect and overwrite
>>    # Redirect and append
|     # Pipeline (pass objects)
2>    # Redirect errors only
*>    # Redirect all streams
```

PowerShell Streams:

1. **Success (1)**: Normal output
2. **Error (2)**: Error messages
3. **Warning (3)**: Warnings
4. **Verbose (4)**: Detailed info
5. **Debug (5)**: Debug output
6. **Information (6)**: General info

Out-File vs Redirection

```powershell
# Redirection operator (simple)
Command > file.txt

# Out-File cmdlet (more control)
Command | Out-File file.txt -Encoding UTF8 -Width 120
```

Task 8: PowerShell Version with Redirection

Objective: Get version info (3 lines) and save to file

Step 1: Find Command that Returns 3 Lines

```powershell
# Test this command
$PSVersionTable.PSVersion
```

Expected Output (3 lines):

```
Major  Minor  Patch  PreReleaseLabel BuildLabel
-----  -----  -----  --------------- ----------
5      1      19041
```

Step 2: Add Output Redirection

```powershell
$PSVersionTable.PSVersion | Out-File "$env:USERPROFILE\Documents\PowerShell.txt"
```

Command Breakdown:

- `$PSVersionTable.PSVersion` → Gets version object
- `|` → Pipes to next command
- `Out-File` → Converts to text and writes file
- `"$env:USERPROFILE\Documents\PowerShell.txt"` → Full file path

📝 **Document**: Complete command including redirection

Verification:

```powershell
# Check file exists
Test-Path "$env:USERPROFILE\Documents\PowerShell.txt"

# View contents
Get-Content "$env:USERPROFILE\Documents\PowerShell.txt"

# File details
Get-Item "$env:USERPROFILE\Documents\PowerShell.txt" | Format-List
```

Exercise 9: File Operations with Absolute Paths

Learning Focus: File movement and absolute path mastery

File Operation Cmdlets

- **Copy-Item**: Copy files/folders
- **Move-Item**: Move/rename files/folders
- **Remove-Item**: Delete files/folders
- **Rename-Item**: Rename files/folders

Move-Item Deep Dive

```powershell
Move-Item [-Path] <String[]> [[-Destination] <String>] [-Force] [-PassThru]
```

Parameters:

- **-Path**: Source file/folder location
- **-Destination**: Target location
- **-Force**: Overwrite existing files
- **-PassThru**: Return moved object

Absolute Path Benefits

- **Unambiguous**: No confusion about location
- **Reliable**: Works regardless of current directory
- **Scriptable**: Consistent in automated processes

Task 9: Move File with Absolute Paths

Objective: Move PowerShell.txt from Documents to Lab4 using absolute paths only

Requirements:

- 1 cmdlet
- 2 arguments (both absolute paths)

Your Command:

```powershell
Move-Item "$env:USERPROFILE\Documents\PowerShell.txt" "$env:USERPROFILE\Documents\Lab4\"
```

Alternative Syntax:

```powershell
Move-Item -Path "$env:USERPROFILE\Documents\PowerShell.txt" -Destination "$env:USERPROFILE\Documents\Lal
```

📝 **Document**: Command using absolute paths

Verification:

```powershell
# File should exist in Lab4
Test-Path "$env:USERPROFILE\Documents\Lab4\PowerShell.txt"

# File should NOT exist in Documents root
Test-Path "$env:USERPROFILE\Documents\PowerShell.txt"
```

Expected Results:

- First test: `True`
- Second test: `False`

Exercise 10: Recursive Directory Listing

Learning Focus: Directory traversal and recursive operations

The Get-ChildItem Cmdlet

```powershell
Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Recurse]
          [-Depth <UInt32>] [-Force] [-Name]
```

Key Parameters:

- **-Recurse**: Include all subdirectories
- **-Depth**: Limit recursion depth
- **-Force**: Show hidden/system files
- **-Name**: Show only names (not full objects)

Recursion Concepts

- **Depth 0**: Current directory only
- **Depth 1**: Current + immediate subdirectories
- **Unlimited**: All subdirectories (default with -Recurse)

Task 10: Recursive Directory Listing

Objective: Show all contents of Documents including subdirectories

Your Command:

```powershell
Get-ChildItem "$env:USERPROFILE\Documents" -Recurse
```

Expected Output Format:

```
    Directory: C:\Users\[USERNAME]\Documents

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        [DATE]   [TIME]                Lab4

    Directory: C:\Users\[USERNAME]\Documents\Lab4

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        [DATE]   [TIME]            xxx PowerShell.txt
```

📸 **Screenshot Required**: Recursive directory listing showing Lab4 and its contents

Enhanced Formatting:

```powershell
# Table format with specific columns
Get-ChildItem "$env:USERPROFILE\Documents" -Recurse | Format-Table Name, Length, LastWriteTime

# List format with more details
Get-ChildItem "$env:USERPROFILE\Documents" -Recurse | Format-List
```

Exercise 11: Relative Path Operations

Learning Focus: Relative path navigation and current directory context

Understanding Current Directory

PowerShell maintains a "current working directory" concept:

- **Get-Location** or **pwd**: Show current directory
- **Set-Location** or **cd**: Change directory
- Relative paths work from this location

Relative Path Symbols

- `.` → Current directory
- `..` → Parent directory
- `.\folder` → Subfolder in current directory
- `..\folder` → Folder in parent directory
- `..\..\folder` → Folder two levels up

Working Directory vs. PowerShell Location

PowerShell can work with various "providers":

- **FileSystem**: Regular files and folders
- **Registry**: Windows Registry
- **Certificate**: Certificate stores
- Current location context matters for relative paths

Task 11: File Copy with Relative Paths

Objective: Copy file using relative paths after changing directory

Requirements:

- 1 cmdlet for copy operation
- 2 arguments (both relative paths)

Step 1: Change to Documents Directory

```powershell
Set-Location "$env:USERPROFILE\Documents"
```

Verification:

```powershell
Get-Location
# Should show: C:\Users\[USERNAME]\Documents
```

Step 2: Copy with Relative Paths

```powershell
Copy-Item ".\Lab4\PowerShell.txt" ".\"
```

Alternative Syntax:

```powershell
Copy-Item "Lab4\PowerShell.txt" "."
```

Understanding the Paths:

- `".\Lab4\PowerShell.txt"` → File in Lab4 subdirectory
- `".\"` or `"."` → Current directory (Documents)

📝 **Document**: Copy command using relative paths

Verification:

```powershell
# List current directory
Get-ChildItem

# Should show PowerShell.txt in both locations:
# - Documents root
# - Lab4 subdirectory
```

Phase 4: Network and System Information

Exercise 12: Network Configuration Retrieval

Learning Focus: Network adapter management and data filtering

Windows Network Architecture

Network Adapter Types:

- **Ethernet**: Wired connections
- **Wi-Fi**: Wireless connections
- **Bluetooth**: Short-range wireless
- **Loopback**: Internal communication (127.0.0.1)
- **Tunnel**: VPN connections

MAC Address Deep Dive

Structure: XX-XX-XX-XX-XX-XX

- **First 3 pairs**: Organizationally Unique Identifier (OUI) - manufacturer
- **Last 3 pairs**: Network Interface Controller specific
- **Purpose**: Unique hardware identifier for network communication

The Get-NetAdapter Cmdlet

```powershell
Get-NetAdapter [[-Name] <String[]>] [-IncludeHidden] [-Physical] [-Virtual]
```

Important Properties:

- **Name**: Human-readable adapter name
- **MacAddress**: Hardware identifier
- **Status**: Up, Down, Disconnected
- **LinkSpeed**: Connection speed
- **MediaType**: 802.3 for Ethernet

Advanced Filtering with Where-Object

```powershell
Where-Object [-FilterScript] <ScriptBlock>
```

Common Comparison Operators:

- `-eq`: Equal to
- `-like`: Wildcard matching (* and ?)
- `-match`: Regular expression
- `-contains`: Collection contains item

Task 12: Network Adapter MAC Addresses

Objective: Get Ethernet adapter MAC addresses, append to file

Requirements:

- Show only Name and MacAddress columns
- 2 columns, 3 lines (header + data lines)
- Append to PowerShell.txt in Lab4

Step 1: Basic Network Adapter Query

```powershell
Get-NetAdapter | Where-Object {$_.Name -like "*Ethernet*"} | Select-Object Name, MacAddress
```

Step 2: With File Append

```powershell
Get-NetAdapter | Where-Object {$_.Name -like "*Ethernet*"} | Select-Object Name, MacAddress >> "$env:USERPR
```

Expected Output:

```
Name            MacAddress
----          ----------
Ethernet        00-0C-29-XX-XX-XX
Ethernet 2       00-0C-29-XX-XX-XX
```

📝 **Document**: Complete command with append redirection

Pipeline Breakdown:

1. `Get-NetAdapter` → Get all network adapters
2. `Where-Object {...}` → Filter for Ethernet adapters
3. `Select-Object Name, MacAddress` → Choose specific columns
4. `>>` → Append to existing file

Alternative Filtering:

```powershell
# Filter by media type
Get-NetAdapter | Where-Object {$_.MediaType -eq "802.3"}

# Filter by status
Get-NetAdapter | Where-Object {$_.Status -eq "Up"}

# Multiple criteria
Get-NetAdapter | Where-Object {$_.Name -like "*Ethernet*" -and $_.Status -eq "Up"}
```

Exercise 13: IP Address Information Pipeline

**