



Příručka ke školení

Automatizace správy pomocí Windows PowerShell



Martin Trnka

Studijní materiály Okškolení

Contents

Module 1: Getting started with Windows PowerShell	
Module Overview	1-1
Lesson 1: Overview and background of Windows PowerShell	1-2
Lab A: Configuring Windows PowerShell	1-12
Lesson 2: Understanding command syntax	1-15
Lesson 3: Finding commands	1-24
Lab B: Finding and running basic commands	1-30
Module Review and Takeaways	1-33
Module 2: Cmdlets for administration	
Module Overview	2-1
Lesson 1: Active Directory administration cmdlets	2-2
Lesson 2: Network configuration cmdlets	2-13
Lesson 3: Other server administration cmdlets	2-19
Lab: Windows administration	2-24
Module Review and Takeaways	2-28
Module 3: Working with the Windows PowerShell pipeline	
Module Overview	3-1
Lesson 1: Understanding the pipeline	3-2
Lesson 2: Selecting, sorting, and measuring objects	3-8
Lab A: Using the pipeline	3-16
Lesson 3: Filtering objects out of the pipeline	3-19
Lab B: Filtering objects	3-25
Lesson 4: Enumerating objects in the pipeline	3-28
Lab C: Enumerating objects	3-32
Lesson 5: Sending pipeline data as output	3-34
Lab D: Sending output to a file	3-39
Module Review and Takeaways	3-41
Module 4: Understanding how the pipeline works	
Module Overview	4-1
Lesson 1: Passing pipeline data	4-2
Lesson 2: Advanced techniques for passing pipeline data	4-8
Lab: Working with pipeline parameter binding	4-13
Module Review and Takeaways	4-16

Module 5: Using PSProviders and PSDrives

Module Overview	5-1
Lesson 1: Using PSProviders	5-2
Lesson 2: Using PSDrives	5-5
Lab: Using PSProviders and PSDrives	5-14
Module Review and Takeaways	5-18

Module 6: Querying management information by using CIM and WMI

Module Overview	6-1
Lesson 1: Understanding CIM and WMI	6-2
Lesson 2: Querying data by using CIM and WMI	6-7
Lesson 3: Making changes by using CIM and WMI	6-15
Lab: Working with CIM and WMI	6-19
Module Review and Takeaways	6-23

Module 7: Working with variables, arrays, and hash tables

Module Overview	7-1
Lesson 1: Using variables	7-2
Lesson 2: Manipulating variables	7-9
Lesson 3: Manipulating arrays and hash tables	7-15
Lab: Working with variables	7-23
Module Review and Takeaways	7-27

Module 8: Basic scripting

Module Overview	8-1
Lesson 1: Introduction to scripting	8-2
Lesson 2: Scripting constructs	8-11
Lesson 3: Importing data from files	8-17
Lab: Basic scripting	8-22
Module Review and Takeaways	8-28

Module 9: Advanced scripting

Module Overview	9-1
Lesson 1: Accepting user input	9-2
Lesson 2: Overview of script documentation	9-9
Lab A: Accepting data from users	9-13
Lesson 3: Troubleshooting and error handling	9-16
Lesson 4: Functions and modules	9-24
Lab B: Implementing functions and modules	9-30
Module Review and Takeaways	9-33

Module 10: Administering remote computers

Module Overview	10-1
Lesson 1: Using basic Windows PowerShell remoting	10-3
Lesson 2: Using advanced Windows PowerShell remoting techniques	10-13
Lab A: Using basic remoting	10-19
Lesson 3: Using PSSessions	10-22
Lab B: Using PSSessions	10-28
Module Review and Takeaways	10-31

Module 11: Using background jobs and scheduled jobs

Module Overview	11-1
Lesson 1: Using background jobs	11-2
Lesson 2: Using scheduled jobs	11-9
Lab: Using background jobs and scheduled jobs	11-16
Module Review and Takeaways	11-20

Module 12: Using advanced Windows PowerShell techniques

Module Overview	12-1
Lesson 1: Creating profile scripts	12-2
Lesson 2: Using advanced techniques	12-5
Lab: Practicing advanced techniques	12-17
Module Review and Takeaways	12-22

Lab Answer Keys

Module 1 Lab A: Configuring Windows PowerShell	L1-1
Module 1 Lab B: Finding and running basic commands	L1-3
Module 2 Lab: Windows administration	L2-9
Module 3 Lab A: Using the pipeline	L3-13
Module 3 Lab B: Filtering objects	L3-17
Module 3 Lab C: Enumerating objects	L3-20
Module 3 Lab D: Sending output to a file	L3-22
Module 4 Lab: Working with pipeline parameter binding	L4-25
Module 5 Lab: Using PSProviders and PSDrives	L5-27
Module 6 Lab: Working with CIM and WMI	L6-31
Module 7 Lab: Working with variables	L7-37
Module 8 Lab: Basic scripting	L8-41
Module 9 Lab A: Accepting data from users	L9-45
Module 9 Lab B: Implementing functions and modules	L9-46

Module 10 Lab A: Using basic remoting	L10-49
Module 10 Lab B: Using PSSessions	L10-52
Module 11 Lab: Using background jobs and scheduled jobs	L11-57
Module 12 Lab: Practicing advanced techniques	L12-51

Studijní materiály Okškolení

About This Course

This section provides a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This course provides students with the fundamental knowledge and skills they need to automate the administration of computers by using Windows PowerShell. Students will learn skills to identify and build commands to perform specific tasks. In addition, they will learn how to build scripts to accomplish advanced tasks such as automating repetitive tasks and generating reports. This course provides prerequisite skills that support a broad range of Microsoft products, including Windows Server, Windows client operating systems, Microsoft Exchange Server, Microsoft SharePoint Server, Microsoft SQL Server, and System Center. In keeping with this goal, the course will not focus on any one of those products. However, Windows Server, which is the common platform for all of those products, will serve as the example for the techniques this course teaches.

Audience

This course is intended for IT pros who are already experienced in general Windows Server and Windows client administration and who want to learn more about using Windows PowerShell for administration. The course doesn't require any prior experience with any version of Windows PowerShell or any scripting language. This course is also suitable for IT pros who are already experienced in server administration, including that for Exchange Server, SharePoint Server, SQL Server, and System Center.

Student Prerequisites

This course requires that you meet the following prerequisites:

- Experience with Windows networking technologies and implementation
- Experience with Windows Server administration, maintenance, and troubleshooting
- Experience with Windows client administration, maintenance, and troubleshooting

Course Objectives

After completing this course, students will be able to:

- Explain the basic concepts behind Windows PowerShell.
- Identify and use basic cmdlets to manage a variety of services by using Windows PowerShell.
- Work with the Windows PowerShell pipeline.
- Describe how the Windows PowerShell pipeline works.
- Use the **PSProviders** and **PSDrives** adapters.
- Use Windows Management Instrumentation (WMI) and Common Information Model (CIM) to manage Windows servers.
- Use variables, arrays, and hash tables in Windows PowerShell.
- Develop basic Windows PowerShell scripts.
- Implement advanced scripting concepts such as gathering input, documenting scripts, and handling errors.
- Administer remote computers.
- Use background jobs and scheduled jobs.
- Use advanced Windows PowerShell techniques and profiles.

Course Outline

The course outline is as follows:

Module 1, "Getting started with Windows PowerShell," introduces students to Windows PowerShell and provides an overview of its functionality. Students will learn to open and configure Windows PowerShell for use and to run commands within it. They will also learn about the Windows PowerShell built-in Help system.

Module 2, "Cmdlets for administration," makes students familiar with the cmdlets that they will use in a production environment. Students can search for cmdlets each time they need to accomplish a task. However, it's more efficient to have at least a basic understanding of the cmdlets available for system administration. The content in this module allows students to improve their understanding of cmdlets, reduce their dependence on the search functionality, and predict naming patterns so that they can find cmdlets easily.

Module 3, "Working with the Windows PowerShell pipeline," introduces the pipeline feature of Windows PowerShell. Although the pipeline feature is part of several command-line shells, such as the command prompt in the Windows operating system, the pipeline feature in Windows PowerShell provides more-complex, more-flexible, and more-capable functionalities compared to other shells. This module provides students with the skills and knowledge to use Windows PowerShell more effectively and efficiently.

Module 4, "Understanding how the pipeline works," explains how the Windows PowerShell command-line interface passes objects from one command to another in the pipeline. Windows PowerShell can use two techniques to pass data: **ByValue** and **ByPropertyName**. By knowing how these techniques work and which one to use in a particular scenario, students can construct more-useful and more-complex command lines.

Module 5, "Using PSProviders and PSDrives," introduces the **PSProviders** and **PSDrives** adapters. A **PSProvider** is a Windows PowerShell adapter that makes a form of storage resemble a hard drive. A **PSDrive** is an actual connection to a form of storage. By using these two technologies, students can work with many forms of storage by using the same commands and techniques that they use to manage the file system.

Module 6, "Querying management information by using CIM and WMI," introduces students to two parallel technologies: WMI and CIM. Both of these technologies provide local and remote access to a repository of management information, including access to robust information available from the operating system, computer hardware, and installed software.

Module 7, "Working with variables, arrays, and hash tables," provides students with the skills and knowledge required to use variables, arrays, and hash tables as one of the steps in learning how to write Windows PowerShell scripts.

Module 8, "Basic scripting," explains how to package Windows PowerShell commands in a script. Scripts allow students to perform repetitive and complex tasks that they can't accomplish in a single command. In addition to learning how to create, run, and modify scripts, students will also learn how to import data from a file.

Module 9, "Advanced scripting," introduces students to more-advanced techniques that they can use in scripts. These techniques includes gathering user input, reading input from files, documenting scripts with help information, and handling errors.

Module 10, "Administering remote computers," introduces students to the Windows PowerShell remoting technology. This technology allows students to connect to one or more remote computers and instruct them to run commands on their behalf. Students will learn how to use remoting to perform administration on remote computers, and they'll establish and manage persistent connections to remote computers.

Module 11, "Using background jobs and scheduled jobs," provides information about the job feature of Windows PowerShell. Jobs are an extension point in Windows PowerShell, and many different kinds of jobs exist. The different kinds of jobs can work slightly differently, and they have different capabilities.

Module 12, "Using advanced Windows PowerShell techniques," introduces some of the advanced techniques and features of Windows PowerShell, including profile scripts, regular expressions, and the format operator. Many of these techniques and features extend the functionality that students learn about in other modules. Some of these techniques are new to Windows PowerShell 5.1 and provide additional capabilities.

Course Materials

The following materials are included with your kit:

- **Course Handbook:** a succinct classroom learning guide that provides the critical technical information in a crisp, tightly-focused format, which is essential for an effective in-class learning experience.
 - **Lessons:** guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
 - **Labs:** provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
 - **Module Reviews and Takeaways:** provide on-the-job reference material to boost knowledge and skills retention.
 - **Lab Answer Keys:** provide step-by-step lab solution guidance.



Additional Reading: Course Companion Content on the <http://www.microsoft.com/learning/en/us/companion-moc.aspx> Site: searchable, easy-to-browse digital content with integrated premium online resources that supplement the Course Handbook.

- **Modules:** include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.
- **Resources:** include well-categorized additional resources that give you immediate access to the most current premium content on TechNet, MSDN, or Microsoft Press.



Additional Reading: Student Course files on the <http://www.microsoft.com/learning/en/us/companion-moc.aspx> Site: includes the Allfiles.exe, a self-extracting executable file that contains all required files for the labs and demonstrations.

- **Course evaluation:** At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
 - To provide additional comments or feedback on the course, send an email to mcspprt@microsoft.com. To inquire about the Microsoft Certification Program, send an email to mcphelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Microsoft Hyper-V to perform the labs.



Note: At the end of each lab, if required, you must revert the virtual machines (VMs) to a checkpoint. You can find the instructions for this procedure at the end of each lab that requires it.

The following table shows the role of each VM that is used in this course.

VM	Role
10961C-LON-DC1	Windows Server 2016 domain controller with the full Desktop Experience
10961C-LON-SVR1	Windows Server 2016 member server with the full Desktop Experience
10961C-LON-CL1	Domain-joined Windows 10 client computer with Remote Server Administration Tools (RSAT) installed

Software Configuration

The following software is installed on each VM:

- Updated Windows PowerShell Help files

The following software is installed on **10961C-LON-CL1**:

- RSAT
- WMI Explorer
- Code signing certificate issued by an internal certification authority

Course Files

The files associated with the labs in this course are located in the **E:\ModXX\Labfiles** folder on the student VMs.

Classroom Setup

Each classroom computer will have the same VMs configured in the same way.

You might access those VMs in a local on-premises classroom or through Microsoft Labs Online:

- On-premises classroom. If you are working on a local computer, you might need to revert the VMs to a checkpoint at the end of each lab. The lab will include the steps to do this.
- Microsoft Labs Online. If you are working in the hosted environment, some variations might exist for the configuration or lab steps in your student manual. The “**Lab Notes**” document on the hosted lab platform will note any differences.

Your Microsoft Certified Trainer will provide more details about your specific lab environment.

Studijní materiály Okškolení

Course Hardware Level

To ensure a satisfactory student experience where labs are being run locally, Microsoft Learning requires a minimum equipment configuration for the trainer and student computers. These requirements are for all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which official Microsoft Learning Product courseware is taught.

Hardware Level 8

- Processor: Minimum of 2.8 gigahertz (GHz) 64-bit processor (multi-core)
 - AMD:
 - Supports AMD Virtualization (AMD-V)
 - Second Level Address Translation (SLAT)—nested page tables
 - Hardware-enforced Data Execution Prevention (DEP) must be available and enabled (NX bit)
 - Supports trusted platform module (TPM) 2.0 or greater
 - Intel:
 - Supports Intel Virtualization Technology (Intel VT)
 - Supports SLAT—Extended Page Tables
 - Hardware-enforced DEP must be available and enabled (XD bit)
 - Supports TPM 2.0 or later
- Hard Disk: 500-gigabyte (GB) solid-state drive (SSD) System Drive with two partitions labeled drive C and drive D
- RAM: Minimum of 32 gigabytes (16 GB may be sufficient for this course)
- Network adapter
- Monitor: Dual monitors supporting a minimum resolution of 1440 x 900
- Mouse or compatible pointing device
- Sound card with headsets

In addition, the trainer computer must:

- Be connected to a projection display device that supports super video graphics adapter (SVGA) 1024 x 768 pixels, 16-bit colors.
- Have a sound card with amplified speakers.

Module 1

Getting started with Windows PowerShell

Contents:

Module Overview	1-1
Lesson 1: Overview and background of Windows PowerShell	1-2
Lab A: Configuring Windows PowerShell	1-12
Lesson 2: Understanding command syntax	1-15
Lesson 3: Finding commands	1-24
Lab B: Finding and running basic commands	1-30
Module Review and Takeaways	1-33

Module Overview

You might be familiar with command-line interfaces from other operating systems or have used the **cmd.exe** shell in Windows. However, you must learn (and perhaps relearn) important concepts and techniques before you can use Windows PowerShell effectively.

This module will introduce you to Windows PowerShell and provide an overview of the product's functionality. You will learn to open and configure Windows PowerShell for use and run commands within it. You will also learn about Windows PowerShell's built-in help system. It plays an important role in helping you learn how to use commands in Windows PowerShell.

 **Additional Reading:** For more information on Windows PowerShell, refer to: "PowerShell Team Blog" at: <https://aka.ms/jwv0e9>

 **Additional Reading:** For community-based help on Windows PowerShell, refer to: "Welcome to PowerShell.org" at: <https://aka.ms/oud2o8>

 **Additional Reading:** For more information from the Official Scripting Guide Forum, refer to: "TechNet" at: <https://aka.ms/ajewak>

Objectives

After completing this module, you will be able to:

- Open and configure Windows PowerShell.
- Discover, learn, and run Windows PowerShell commands.
- Find Windows PowerShell commands for performing specific tasks.

Lesson 1

Overview and background of Windows PowerShell

It is easy to simply begin using Windows PowerShell without learning about its background first. However, you use Windows PowerShell more easily and effectively by understanding its background and intended use.

In this lesson, you will learn about Windows PowerShell's system requirements. You will also learn to open and configure the two included host applications—the Windows PowerShell console and Windows PowerShell Integrated Scripting Environment (ISE).

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of Windows PowerShell.
- List the major versions of Windows PowerShell.
- Describe the differences between shell features and operating system features.
- Describe considerations when using multiple versions of Windows PowerShell.
- Identify the two native Windows PowerShell hosting applications.
- Display Windows PowerShell version information.
- Describe the precautions you can take to avoid mistakes when opening Windows PowerShell.
- Explain how to configure the Windows PowerShell console host.
- Configure the Windows PowerShell console host.
- Explain how to configure the Windows PowerShell ISE host.
- Configure the Windows PowerShell ISE host.

Windows PowerShell overview

Introduced in 2006, Windows PowerShell is a platform built on Microsoft .NET. You can use Windows PowerShell to automate administrative tasks. Because it is built on .NET, Windows PowerShell is object-oriented. This provides more power and flexibility when working with data as compared to working with all data simply as text. You also have access to the same functionality that the .NET Framework provides and to other code libraries that software developers use.

Commands provide Windows PowerShell's main functionality. There are many varieties of commands, including cmdlets (pronounced *command-lets*), functions, filters, scripts, applications, configurations, and workflows. Commands are building blocks that you piece together by using the Windows PowerShell scripting language. By using commands, you can create custom solutions for complex administrative problems. Alternatively, you can simply run commands directly within the Windows PowerShell console to complete a single task. The console is the command-line interface (CLI) for Windows PowerShell and is the primary way in which you will interact with Windows PowerShell.

- Introduced in 2006
- Both a CLI and scripting language
- Commands include:
 - Cmdlets (pronounced *command-lets*)
 - Functions
 - Filters
 - Workflows
- Microsoft server applications provide application-specific cmdlets
- Now open-source and supports Linux and macOS

Windows PowerShell replaces the previous **cmd.exe** CLI and the limited functionality of its batch file scripting language.

Microsoft server applications and cloud services provide specialized cmdlets that you can use to manage those services. In fact, you can manage some features only by using Windows PowerShell. In many cases, even when the application provides a graphical user interface (GUI) to manage the software, it is hosting Windows PowerShell and running the cmdlets behind the scenes.

Applications and services with Windows PowerShell-based administrative functions are consistent in how they work. This means that you can quickly apply the lessons you learned. Also, when you use automation scripts to administer a software application, you can reuse them among other applications as well.

Starting in 2016, Windows PowerShell became an open-source project, currently called PowerShell Core, and Microsoft enabled support for running Windows PowerShell on Linux. This extends the usefulness of Windows PowerShell beyond Windows administrative environments. You can use Windows PowerShell to manage:

- Linux servers and macOS from Windows Server.
- Windows Server from Linux and macOS.

Windows PowerShell versions

Windows PowerShell has five versions: Windows PowerShell 5.0, Windows PowerShell 4.0, Windows PowerShell 3.0, Windows PowerShell 2.0, and Windows PowerShell 1.0. Windows PowerShell 5.0 is backward-compatible with the older versions of Windows PowerShell, except version 1.0. Be aware that many server products have a dependency on a specific version of Windows PowerShell. For example, Microsoft Exchange Server 2010 requires Windows PowerShell 2.0, and is not compatible with newer versions of Windows PowerShell.

Before installing a new version of Windows PowerShell on a computer, verify the compatibility of that version with the software installed on that computer.

	2.0	3.0	4.0	5.1
Windows XP	Available	No	No	No
Windows Server 2003	Available	No	No	No
Windows Vista	Available	No	No	No
Windows Server 2008	Available	Available with SP2	No	No
Windows 7	Installed	Available with SP1	Available	Available
Windows Server 2008 R2	Installed	Available with SP2	Available	Available
Windows 8	No	Installed	Available	Available
Windows Server 2012	No	Installed	Available	Available
Windows 8.1 and Windows Server 2012 R2	No	No	Installed	Available
Windows 10 Anniversary Edition and Windows Server 2016	No	No	No	Installed

Windows PowerShell 1.0

Windows PowerShell 1.0 was the initial release, and it was available as an installation option in Windows Vista and Windows Server 2008. Currently, no applications or server products have a firm dependency on Windows PowerShell 1.0, and most computers should be fine having at least Windows PowerShell 2.0 installed. Windows PowerShell 1.0 was compatible with Windows Vista, Windows Server 2008, Windows XP, and Windows Server 2003.

Windows PowerShell 2.0

Windows PowerShell 2.0 was a default installation on Windows 7 and Windows Server 2008 R2. It was available as a free download for Windows Vista, Windows Server 2008, Windows XP, and Windows Server 2003. The download package is the Windows Management Framework. This also includes Windows Remote Management (WinRM), Background Intelligent Transfer Service (BITS), and other components. Windows PowerShell 2.0 is generally backward-compatible with Windows PowerShell 1.0.

Windows PowerShell 3.0

Windows PowerShell 3.0 is a default installation on Windows 8 and Windows Server 2012. It is also available as an out-of-band free download for Windows 7 with Service Pack 1 (SP1), Windows Server 2008 SP2, and Windows Server 2008 R2 SP1. Be aware that Windows PowerShell 3.0 is incompatible with Windows XP, Windows Server 2003, and Windows Vista. Windows Management Framework 3.0 is the out-of-band free download package in which Windows PowerShell 3.0 is available. It also includes Windows Management Instrumentation (WMI), WinRM, Open Data Protocol (OData), Internet Information Services (IIS) Extension, and Server Manager Common Information Model (CIM) Provider updates. The WinRM feature in Windows PowerShell 3.0 is compatible with the same feature in Windows PowerShell 2.0. Therefore, the two versions can successfully communicate with one another for management.

Windows PowerShell 3.0 requires version 4.0 of the .NET Framework. Older versions of Windows PowerShell required version 2.0 of the .NET Framework.

Windows PowerShell 4.0

Windows PowerShell 4.0 is a default installation on Windows 8.1 and Windows Server 2012 R2. It is also available as an out-of-band free download for Windows 7 with Service Pack 1 SP1, Windows Server 2008 R2 SP1, Windows 8, and Windows Server 2012. Windows PowerShell 4.0 is not compatible with earlier versions of Windows. Windows Management Framework 4.0 is the out-of-band free download package in which Windows PowerShell 4.0 is available. Windows PowerShell 4.0 requires version 4.5 of the .NET Framework.

Windows PowerShell 5.0

Windows PowerShell 5.0 is a default installation on Windows 10 Version 1511. You can install Windows PowerShell 5.0 on Windows Server 2008 R2, Windows Server 2012, Windows 10, Windows 8.1 Enterprise, Windows 8.1 Pro, or Windows 7 SP1. To do so, you can download and install the Windows Management Framework 5.0 from the Windows Download Center.

Windows PowerShell 5.1

Windows PowerShell 5.1 released as part of Windows Management Framework (WMF) 5.1 and is included in Windows Server 2016 and Windows 10 Anniversary Edition. You can install Windows PowerShell 5.1 on all the operating systems that you can install Windows PowerShell 5.0 on. To do so, you can download and install the Windows Management Framework 5.1 from the Windows Download Center.



Note: To install Windows Management Framework 5.0 on Windows Server 2008 and Windows 7, you must have Windows Management Framework 4.0 and version 4.5 of the .NET Framework installed. Windows Management Framework 5.1 requires that you install .NET Framework 4.5.2. Installation may succeed, but some features will fail if you have not installed .NET Framework 4.5.2.



Additional Reading: For more information about installing and configuring WMF 5.1, refer to: "Install and Configure WMF 5.1" at: <https://aka.ms/u33mya>



Additional Reading: For more information on the Windows Management Framework 5.1, refer to: "Microsoft Download Center" at: <https://aka.ms/i31ojp>

Windows PowerShell vs. operating system

Each version of Windows PowerShell includes certain core, native functionality. This functionality is available regardless of the operating system version where Windows PowerShell is running. For example, Windows PowerShell 5.0 introduces the **New-TemporaryFile** cmdlet, and that cmdlet is available whether Windows PowerShell is running on Windows 10, Windows 8, or a server operating system.

Much of Windows PowerShell's usefulness stems from its extensibility. However, Windows PowerShell by itself does not include extensions.

For example, Windows Server 2016 includes cmdlets to manage Domain Name System (DNS) Response Rate Limiting (RRL). You can import the DNS Server module that contains the cmdlets. However, because RRL is a new feature for Windows Server 2016, those cmdlets will only work on Windows Server 2016. In other words, one advantage of moving to the newer operating system versions is that they provide more support for management based on Windows PowerShell.

- Windows PowerShell ships with specific core, native functionality
- Most of its use, however, comes from extensions—additional commands that extend the capabilities of Windows PowerShell
- Extensions are designed to work with a specific version of Windows PowerShell, but they do not ship with Windows PowerShell itself
- Instead, extensions are provided as part of an operating system or a specific product version

Two host applications

To interact with Windows PowerShell, you must use an application that embeds, or *hosts*, Windows PowerShell's engine. Some of those applications will be GUIs, such as the Exchange Management Console in Exchange Server 2007 and newer versions. In this course, you will primarily interact with Windows PowerShell through one of the two CLI hosts that Microsoft provides: the Windows PowerShell console and the Windows PowerShell ISE.

- | |
|---|
| <ul style="list-style-type: none"> • Console <ul style="list-style-type: none"> • Basic command-line interface • Maximum support for PowerShell features • Minimal editing capabilities • ISE <ul style="list-style-type: none"> • Script editor and console combination • Some Windows PowerShell features not supported • Rich editing capabilities • Third-party hosting applications/editors <ul style="list-style-type: none"> • Varying features and pricing |
|---|

Windows PowerShell console

The console uses the Windows built-in console host application. This is basically the same command-line experience that the older **cmd.exe** shell offered. However, Windows Server 2016 and Windows 10 provide an updated console with more functionality, including coloring of syntax. The console provides a straightforward environment, does not support double-byte character sets, and has limited editing capabilities. It currently provides the broadest Windows PowerShell functionality. The console in Windows PowerShell 5.1 includes significant improvements.

Windows PowerShell ISE

The ISE is a Windows Presentation Foundation (WPF) application that provides rich editing capabilities, IntelliSense code hinting and completion, and support for double-byte character sets. In this course, you will begin by using the console. This is a good way to reinforce important foundational skills. Toward the end of the course, you will shift to using the ISE as you begin to link multiple commands together into scripts. Your instructor might use the ISE in demonstrations, as the ISE makes it easier to run the demonstration scripts supplied with this course.

Studijní materiály pro skolení

Third parties can provide other Windows PowerShell host applications. Several companies produce free and commercial Windows PowerShell scripting, editing, and console hosts. This course will focus on the host applications provided with the Windows operating system.

Working in mixed-version environments

Many organizations will have computers running several different versions of Windows PowerShell. Sometimes, these mixed-version environments are the result of organizational policies that do not permit the installation of newer versions of Windows PowerShell. They might also be the result of software products, especially server software, that have a dependency on a specific version of Windows PowerShell.

What version are you running?

When you are viewing a Windows PowerShell session, it can be difficult to determine which version you are using. All versions of Windows PowerShell install to **%systemdir%\WindowsPowerShell\v1.0**. This refers to the *language* of Windows PowerShell 1.0. This means that the folder name is not helpful in verifying the version number of Windows PowerShell itself.



Note: Windows PowerShell 5.0, Windows PowerShell 4.0, Windows PowerShell 3.0, Windows PowerShell 2.0, and Windows PowerShell 1.0 use the same language version. The language version indicates the keywords used for Windows PowerShell scripts.

To correctly determine the version, type **\$PSVersionTable** in Windows PowerShell, and then press Enter. Windows PowerShell will display the version numbers for various components. This includes the main Windows PowerShell version number. Be aware that this technique will not work in Windows PowerShell 1.0. Instead, it will return a blank result.

Managing version issues

The best way to avoid version problems, of course, is to install the latest version of Windows PowerShell on each machine where you need to run cmdlets. You can deploy the latest version of Windows PowerShell by using Group Policy or through Desired State Configuration (DSC). You may also see automation scripts that are designed to run on earlier versions of Windows PowerShell.

For example, Windows PowerShell 2.0 is available on all systems since Windows 7 and Windows Server 2008 R2. When you install Windows PowerShell 3.0 or newer on older systems that already have Windows PowerShell 2.0, the new version of Windows PowerShell will install side by side, leaving the Windows PowerShell 2.0 engine available for execution. The Windows PowerShell 2.0 engine is also available on operating systems that ship with Windows PowerShell 3.0 or newer. It is possible to force scripts to use the Windows PowerShell 2.0 engine. To switch Windows PowerShell into using the Windows PowerShell 2.0 engine, in the console, type the following command, and then press Enter:

```
PowerShell.exe -Version 2
```

Keep in mind that running Windows PowerShell in 2.0 mode does not provide 100 percent Windows PowerShell 2.0 compatibility, because the behavior of some commands has changed in newer versions. Running in 2.0 mode changes only the version of the Windows PowerShell engine used to run those commands. It is also important to keep in mind that by using an older version of Windows PowerShell's engine, you are bypassing any security features that the newer versions of Windows PowerShell include.

A better option, if you cannot install the latest version of Windows PowerShell on all systems, is to check the version of Windows PowerShell installed on a computer by using the **\$PSVersionTable** command. You can then change your script logic or the commands you plan to use based on that information.

 **Note:** Because so many organizations operate a mixed-version environment, this course attempts to be as version-neutral as possible. Whenever a topic or syntax differs from one version to another, this course will include a note so that the difference is clear.

Precautions when opening Windows PowerShell

On 64-bit operating systems, the Windows PowerShell host applications are available in both 64-bit (x64) and 32-bit (x86) versions. Typically, you will use the 64-bit version that displays as **Windows PowerShell** or **Windows PowerShell ISE** in the Start menu. The 32-bit versions are provided for compatibility with locally installed 32-bit shell extensions and display as **Windows PowerShell (x86)** or **Windows PowerShell ISE (x86)** in the Start menu. As soon as you open Windows PowerShell, the application window title bars reflect the same names as those in the Start menu. Be certain that you are opening the appropriate version for the task you want to perform.

On 32-bit operating systems, Windows PowerShell's host applications are available only in 32-bit versions, and the icons and window title bars do not carry the (x86) designation. Instead, they display simply as **Windows PowerShell** and **Windows PowerShell ISE** in the Start menu.

On computers that have User Account Control (UAC) enabled, you can open Windows PowerShell without administrative credentials. Because you will use Windows PowerShell to perform frequent administrative tasks, you might have to confirm that it opens with full administrative credentials. To do this, right-click the application icon, and then select **Run as Administrator**. When you are running Windows PowerShell with administrative credentials, the application's window title bar will display **Administrator**. Make sure that you check this when you open Windows PowerShell.

- 64-bit operating systems include 64-bit and 32-bit versions
 - 32-bit versions carry **(x86)** designation on icon and window title bar
 - Be certain you are opening the appropriate version for the task at hand
 - Usually, open the 64-bit version if it is available
- Ensure that the window title bar displays **Administrator** if you need administrative privileges in Windows PowerShell
 - When UAC is enabled, you must right-click the application icon to run as Administrator
 - Always verify the window title bar contents when opening Windows PowerShell

Configuring the console

When you work in the console regularly, you might want to customize the environment. You can manage your preferences by changing components such as font size and type, adjusting the size and position of the console window and the screen buffer, changing the color scheme, and enabling keyboard shortcuts.

Font type, color, and size

If the font size and color make it difficult for you to read, you can change them. You might also find that the font style and size that are set in the console make it difficult to differentiate between important characters. This can make it challenging to troubleshoot long, complex lines of syntax. When you first open the console, type the following characters and make sure that you can easily differentiate between them:

- Grave accent (`)
- Single quotation mark (`)
- Open parentheses `(`)
- Open curly bracket `{`}
- Open square bracket `[`
- Open angle bracket or greater than symbol `(>)`

If it is hard to see the difference between these characters, try changing the font. To change the font, select the **Windows PowerShell** icon in the upper-left corner of the console window. From the shortcut menu, select **Properties**, and then select the **Font** tab. Select a new font style and size. TrueType fonts, indicated by the double-T symbol, are usually easier to read than Raster fonts. The Windows PowerShell **Properties** dialog box provides a preview pane in which you can see the results of your choice.

Screen buffer and console window sizes

You can customize the size and location of the console window. When setting the window size, you should understand the relationship between the **Screen Buffer Size** and **Windows Size** options, so that you do not accidentally miss any of the output that appears on the console window. **Screen Buffer Size** is the width in number of characters, and the height in number of lines that appear in your text buffer. The **Windows Size** option, as the name suggests, is the width of the actual window. In most cases, you will want to make sure that the **Width** option for **Screen Buffer Size** is equal to or smaller than the **Width** option for **Windows Size**. This will prevent you from having a horizontal scroll bar on your console window. You can update these options in the **Layout** tab.

Set the width of both the screen buffer and console windows to be as close to the width of your actual screen as possible, to maximize the amount of horizontal text you can display. This will prove useful later when you format the on-screen output of data that your scripts return.

The height of your screen buffer does not have to match your screen height. In most cases, it is much larger. Setting this value to a large number gives you the ability to scroll through large numbers of commands entered in a session or through large amounts of output.

In the **Layout** tab, you can also set a specific location where the console window appears on screen. To do this, set the values for the location in which the top-left corner of the window should appear.

- Select a font style, size, and color, and set the screen color so that text is easy to read and you can differentiate between often-confused characters, such as:
 - ` ` { [<
- Modify screen size to maximize available space for output
- Make sure that the screen buffer width is smaller than window width
- Enable copy and paste

Color scheme

Finally, if you want to change the default white-on-blue color scheme, you can select from a small range of alternative colors in the console's **Properties** dialog box. Use the **Colors** tab to do this. You can change only the primary text and the background color for the main window and any secondary popup windows. You cannot change the color of error messages or other output from this dialog box.

After updating settings, close the dialog box and verify that the window fits on the screen and that no horizontal scroll bar displays. A vertical scroll bar will still appear in the console.

Copying and pasting

The console host supports copying and pasting to and from the clipboard. It also supports the use of standard keyboard shortcuts, though they might not always work. This could be due to other applications that are running on the local computer and have changed settings for keyboard shortcuts. To enable this functionality, make sure that **QuickEdit Mode** is enabled in the console's **Properties** dialog box. In the **Edit Options** section of the **Options** tab, enable keyboard shortcuts for copy and paste by selecting **Enable Ctrl key shortcuts**.

Within the console, select a block of text, and then press Enter to copy that text to the clipboard. Right-click to paste. In Windows 10 and Windows Server 2016, the Ctrl+C and Ctrl+V shortcuts also work for copy and paste.

 **Note:** The Windows console in Windows Server 2016 contains a variety of updates, including support for new keyboard shortcuts, windows resizing, and selecting text.

 **Additional Reading:** For a list of updates to the Windows Console in Windows Server 2016, refer to: "What's New in the Windows Console in Windows Server 2016 Technical Preview" at: <https://aka.ms/ebr7o4>

Demonstration: Configuring the console

In this demonstration, you will see how to:

- Run the 64-bit console as Administrator.
- Set a font family.
- Set a console layout.
- Start a transcript.

Demonstration Steps

1. Open the 64-bit Windows PowerShell console as **Administrator**.
2. Open the **Properties** dialog box of the console host.
3. Select the **Consolas** font and a suitable font size.
4. Configure the window layout so that the entire window fits on the screen and does not display a horizontal scroll bar.
5. Start a shell transcript.
6. Run the **Get-ChildItem** cmdlet.
7. Copy the output into Notepad.

8. Use the Up arrow key to display the previously run command.
9. Close Windows PowerShell.
10. Open the transcript file **C:\Day1.txt**.
11. Close all open windows. Do not save changes in Notepad.

Configuring the ISE

The ISE is a fully graphical environment that provides a script editor, debugging capabilities, an interactive console, and several tools that help you discover and learn new Windows PowerShell commands. This module provides a basic familiarity with how the ISE works. Later modules explain and demonstrate many of the ISE's capabilities more fully.

- Two panes: script and console
- One-pane and two-pane view options
- **Command Add-on** displays available commands
- Customization of font style, size, and color
- Customization of screen color
- Bundling of color selections into themes
- Additional features include snippets, add-ins, and debugging

Panes

The ISE offers two main panes: a Script pane (or script editor) and the Console pane. You can position these one above the other or side-by-side in a two-pane view. You can also maximize one pane and switch back and forth between the panes. By default, a Command Add-on pane also displays, which enables you to search for or browse available commands, and view and fill in parameters for a command you select. There is also a floating **Command** window that provides the same functionality.

Customizing the view

The ISE provides several ways to customize the view. A slider in the lower-right area of the window changes the active font size. The **Options** dialog box lets you customize font and color selection for many different Windows PowerShell text elements, such as keywords and string values. The ISE supports the creation of visual themes. A *theme* is a collection of font and color settings that you can apply as a group to customize the appearance of the ISE. There are several built-in themes that package customizations for purposes such as giving presentations. The ISE also gives you the option to create custom themes.

Other ISE features include:

- A built-in, extensible snippets library that you can use to store commonly used commands.
- The ability to load add-ins created by Microsoft or by third parties that provide additional functionality.
- Integration with Windows PowerShell's debugging capabilities.

Demonstration: Configuring the ISE

In this demonstration, you will see how to:

- Run the ISE as Administrator.
- Configure the pane layout.
- Dock and undock the command pane.
- Configure the font size.
- Select a color theme.

Demonstration Steps

1. On **LON-CL1**, use the Windows PowerShell taskbar icon to open the Windows PowerShell ISE as Administrator.
2. Use toolbar buttons to arrange the **Script** pane and **Console** pane.
3. Open the **Command** add-on and the **Command** window.
4. Change the font size.
5. Select a color theme.
6. Close the Windows PowerShell ISE.

Question: Why might you decide to use the ISE over the console host?

Lab A: Configuring Windows PowerShell

Scenario

You are an administrator who will use Windows PowerShell to automate many administrative tasks. You must make sure that you can successfully start the correct Windows PowerShell host applications, and configure those applications for future use by customizing their appearance.

Objectives

After completing this lab, you will be able to:

- Open and configure the Windows PowerShell console application.
- Open and configure the Windows PowerShell ISE application.

Lab Setup

Estimated Time: **15 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **ADATUM\Administrator**

Password: **Pa55w.rd**

 **Note:** Be aware that you will lose the changes you make during this lab if you revert your VMs at another time during the course.

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Configuring the Windows PowerShell console application

Scenario

To customize Windows PowerShell, you need to first make changes to the console. In this exercise, you will open the Windows PowerShell console application and configure its appearance and layout.

The main tasks for this exercise are as follows:

1. Start the console application as Administrator and pin the Windows PowerShell icon to the taskbar.
2. Configure the Windows PowerShell console application.
3. Start a shell transcript.

► **Task 1: Start the console application as Administrator and pin the Windows PowerShell icon to the taskbar**

1. On **LON-CL1**, start the **Windows PowerShell** application as **Administrator**. Make sure that the window title bar reads Administrator and does not include the text (x86). This indicates that it is the 64-bit console application and that an administrator is running it.
2. Pin the Windows PowerShell icon to the taskbar.

► **Task 2: Configure the Windows PowerShell console application**

1. Open **Windows PowerShell Properties** and configure Windows PowerShell to use the **Consolas** font with **16**-point size.
2. From the **Colors** tab, select alternate display colors for primary text and background.
3. From the **Layout** tab, size the window to fit on the screen and to remove any horizontal scroll bar.

► **Task 3: Start a shell transcript**

- Start a transcript named **C:\DayOne.txt** by running:

```
Start-Transcript C:\DayOne.txt
```

Results: After completing this exercise, you will have opened and configured the Windows PowerShell console application and configured its appearance and layout.

Exercise 2: Configuring the Windows PowerShell ISE application

Scenario

In this exercise, you will customize the appearance of the Windows PowerShell ISE application.

The main tasks for this exercise are as follows:

1. Open the Windows PowerShell ISE application as Administrator.
2. Customize the appearance of the ISE to use the single-pane view, hide the Command pane, and adjust the font size.

► **Task 1: Open the Windows PowerShell ISE application as Administrator**

- Right-click the **Windows PowerShell** icon on the taskbar and open the **Windows PowerShell ISE** application as **Administrator**.

► **Task 2: Customize the appearance of the ISE to use the single-pane view, hide the Command pane, and adjust the font size**

1. Configure the ISE to use the single-pane view and display the console pane.
2. Hide the **Command** pane.
3. Adjust the font size so that you can read it comfortably.
4. Close the **Windows PowerShell ISE** and the **Windows PowerShell** windows.

► **Task 3: Prepare for the next lab**

- Leave the virtual machines running for the next lab.

Results: After completing this exercise, you will have customized the appearance of the Windows PowerShell Integrated Scripting Environment (ISE) application.

Question: Why might you configure alternative text colors in the ISE?

Question: What causes a horizontal scroll bar in the Windows PowerShell console window?

Lesson 2

Understanding command syntax

In this lesson, you will learn about the syntax for using Windows PowerShell cmdlets. You will also learn how to use **Get-Help** to retrieve detailed information about a cmdlet and its parameters.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe cmdlet structure.
- Identify how to use Windows PowerShell parameters.
- Explain how to use tab completion.
- Explain how to use **Get-Help**.
- View help.
- Explain how to interpret the help file contents.
- Explain how to update the local help content.
- Explain how to display the About file content.
- Use About files.

Cmdlet structure

There are thousands of Windows PowerShell cmdlets built into the Windows operating systems and other Microsoft products. Memorizing the names and the syntax for all these commands would be virtually impossible. Fortunately, cmdlet creators build cmdlets by using a common structure that helps you predict both the name of a cmdlet and the syntax. This makes it much easier to discover and use cmdlets.

 **Note:** The basic structure of a cmdlet name is *Verb-Noun*.

Verb is the action the cmdlet performs:

- **Get**
- **Set**
- **New**
- **Add**
- **Remove**

Noun is the resource the cmdlet affects:

- **Service**
- **Process**

Prefixes used to group related nouns:

- AD, SP, and AzureAD

Cmdlet verbs

The verb portion of the cmdlet name indicates what the cmdlet does. There is a set of approved verbs that cmdlet creators use, which provides consistency in cmdlet names. Common verbs include:

- **Get**. Retrieves a resource, such as a file or a user.
- **Set**. Changes the data associated with a resource, such as a file or user property.
- **New**. Creates a resource, such as a file or user.
- **Add**. Adds a resource to a container of multiple resources.
- **Remove**. Deletes a resource from a container of multiple resources.

These represent just some of the verbs that cmdlets use. In addition, some verbs perform similar functions. For example, the **Add** verb can create a resource, similar to the **New** verb. Some verbs might seem similar but have very different functions. For example, the **Read** verb gets information contained in a resource, such as the content of a text file, whereas the **Get** verb retrieves the actual file.

Cmdlet nouns

The noun portion of the cmdlet name indicates what kinds of resources or objects the cmdlet affects. All cmdlets that operate on the same resource should use the same noun. For example, the **Service** noun is used for cmdlets that work with Windows services and the **Process** noun is used for managing processes on a computer.

Nouns can also have prefixes that help the grouping of related nouns into families. For example, the Active Directory nouns start with the letters **AD** (such as **ADUser**, **ADGroup**, and **ADComputer**). Microsoft SharePoint Server cmdlets begin with the prefix **SP**, and Microsoft Azure Active Directory cmdlets begin with the prefix **AzureAD**.



Note: Design guidelines for cmdlets specify that the noun should be a singular noun.



Note: Windows PowerShell uses the generic term *command* to refer to cmdlets, functions, workflows, applications, and other items. These items differ in terms of creation method, but for now you can consider them to all work in the same way. This module uses the terms *command* and *cmdlet* interchangeably.

Parameters

Parameters modify the actions that a cmdlet performs. Each cmdlet can have no parameters, one parameter, or many parameters.

Parameter format

Parameter names begin with a dash (-). A space separates the value that you want to pass from the parameter name. If the value that you are passing contains spaces, then you will need to wrap the text in quotation marks. Some parameters accept multiple values, which are separated by commas, and no spaces.

- Parameters modify the action of a cmdlet
- Names are entered starting with a dash (-)
- Parameters can be optional or required
 - You will receive prompts for required parameters, if needed
- Some accept multiple values, separated by commas
- Parameter name is optional for positional parameters

Optional vs. required parameters

Parameters can be optional or required. If a parameter is required and you run the cmdlet without providing a value for that parameter, Windows PowerShell will prompt you to provide a value for that parameter. For example, if you run the command **Get-Item**, you will receive the following message from Windows PowerShell, which includes a prompt to provide a value for the **-Path** parameter:

```
PS C:\Windows\system32> Get-Item
cmdlet Get-Item at command pipeline position 1
Supply values for the following parameters:
Path[0]:
```

If you type the text **C:** at the prompt and press Enter twice, the command will run successfully. You must press Enter twice because this parameter can accept multiple values. Windows PowerShell will keep prompting you for individual entries until you stop providing them and press Enter.

In some cases, typing the parameter name is optional and you can just enter the value for the parameter. If you run the command **Get-ChildItem C:**, it is the same as running the command **Get-ChildItem -Path C:** because the parameter **-Path** is defined as the first parameter in the cmdlet definition. This is known as a positional parameter and you will see these throughout this course. Omitting the parameter name only works in case a parameter position has been defined. Not all commands have positional parameters.

Switches

Switches are a special case for parameters. Switches are basically parameters that accept a Boolean value (**true** or **false**). They differ from actual Boolean parameters in that the value is only set to **True** if the switch is included when running the command. An example is the **-Recurse** parameter or switch of the **Get-ChildItem** cmdlet. The command **Get-ChildItem c:\ -Recurse** will return not just the items in the C:\ directory, but also those in the subdirectories. Without the **-Recurse** switch, only the items in the C:\ directory are returned.

Tab completion

Tab completion is a Windows PowerShell feature that improves the speed and accuracy of finding and entering cmdlets and parameters. Type a few characters of a cmdlet in the ISE or console and press the Tab key. Windows PowerShell will automatically complete the cmdlet name for you. If there are multiple cmdlets that you could be using, just press TAB multiple times until the desired name appears. This works for cmdlets and parameters, in addition to variable names, object properties, and file paths.

- Allows you to enter cmdlet, parameter, variable, and path names more quickly and accurately
- Helps you to discover cmdlets and parameters
- Supports the use of wildcards

Improving speed and accuracy

Tab completion enables you to enter commands much faster and makes your code less prone to errors. Some cmdlet names can be lengthy and complicated. For example, you might accidentally reverse just two letters when typing the cmdlet name **Get-DnsServerResponseRateLimitingExceptionList**. This would lead to frustration when the command failed to run successfully.

Discovering cmdlet and parameter names

Tab completion also helps you discover cmdlet and parameter names. For example, if you know that you want a **Get** cmdlet that works on an Active Directory resource, you can enter the text **Get-AD** in the console and press TAB to view the various options. For parameters, just enter a dash (-) and you can press the Tab key multiple times to view all the parameters for a cmdlet.

Tab completion even works with wildcards. If you know you want a cmdlet that operates on services, but are not sure which one you want, you can enter the text ***-service** in the console and press TAB to view all the cmdlets that contain the text **-service** in their names.

Using Get-Help

Windows PowerShell provides extensive in-product help for commands. You can access this help by using the **Get-Help** command. **Get-Help** displays all help content on the screen and lets you scroll backward through the content. You can also use the **Help** function or the **Man** alias, which are shortcut functions that map to the **Get-Help** command. All three return basically the same results on-screen, which include a short and long description of the cmdlet, the syntax, any additional remarks from the help author, and even links to related cmdlets or additional information online. The **help** and **Man** commands display content in a page-at-a-time format, in the console. In the ISE they display all the content at one time.

- Displays Windows PowerShell help content
- You provide a cmdlet name to display help for a cmdlet
- Supports wildcards
- Parameters include:
 - *-Examples*
 - *-Full*
 - *-Online*
 - *-ShowWindow*
 - *-Parameter ParameterName*

For example, to display the help information for the **Get-ChildItem** cmdlet, type the following command in the console, and then press Enter:

```
Get-Help Get-ChildItem
```

Get-Help parameters

The **Get-Help** command accepts parameters that help you find additional information beyond the information displayed by default. One of the most common reasons to seek out help is to find usage examples for a command. Windows PowerShell commands typically include many such examples. For instance, run the command **Get-Help Stop-Process -Examples** to see examples of using the **Stop-Process** cmdlet.

The *-Full* parameter provides in-depth information about a cmdlet, including:

- A description of each parameter.
- Whether each parameter has a default value (although this information is not consistently documented across all commands).
- Whether a parameter is mandatory.
- Whether a parameter can accept a value in a specific position (in which case the position number, starting from 1, is given), or whether you must type the parameter name (in which case **named** is shown).
- Whether a parameter accepts pipeline input, and if this is the case, how (which will become important in the next module).

Other **Get-Help** parameters include:

- *-ShowWindow*. Displays the help topic in a separate window, which makes it much easier to view help while entering commands.
- *-Online*. Displays the online version of the help topic (typically the most up to date) in a browser window.
- *-Parameter ParameterName*. Displays the description of a named parameter.
- *-Category*. Displays help only for certain categories of commands, such as cmdlets and functions.

Using Get-Help to find commands

The **Get-Help** command can be very useful for finding commands. It accepts wildcard characters, notably the asterisk (*) wildcard character. When you ask for help and use wildcard characters with a partial command name, Windows PowerShell will display a list of matching help topics.

By using the information you learned earlier about the verb-noun structure of cmdlets, you can use **Get-Help** as a tool to discover cmdlets even if you do not know their names. For example, if you want all the cmdlets that operate on processes, you can type the command **Get-Help *process*** in the console, and then press Enter. The results are like the results for the command **Get-Command *process***, except that **Get-Help** displays the *synopsis*, a short description that helps you identify the command you want.

Sometimes, you might specify a wildcard search that does not match any command name. For example, running **Get-Help *beep*** will not find any commands that have *beep* in their name. When it cannot find any results, the help system performs a full-text search of available command descriptions and synopses. This would locate any help files that contain *beep*. If there is only a single file with a match, the help system will display it instead of showing a one-item list. In the case of *beep*, **Get-Help** returns a list of two topics: **Set-PSReadlineOption**, a cmdlet, and **about_Special_Characters**, a conceptual help topic.

Demonstration: Viewing help

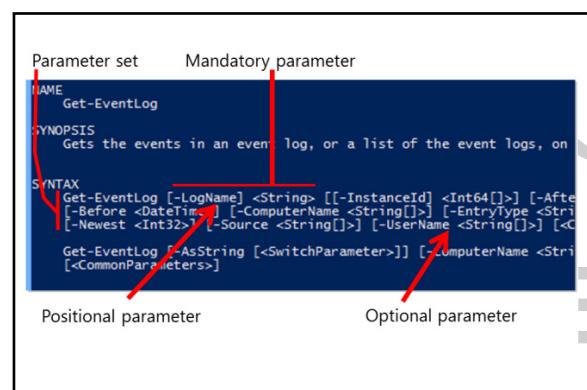
In this demonstration, you will see how to use various options of the help system.

Demonstration Steps

1. Display basic help for a command.
2. Display help in a floating window.
3. Display usage examples.
4. If connected to the Internet, display online help.

Interpreting the help syntax

When you find the command that you need for a task, you can use its help file to learn how to use it. One way to do this is to look at the examples and try to understand the values used. However, it is uncommon for examples to provide all the possible variations of use for a command. Learning to interpret the help file syntax can enable you to decipher a command's capabilities quickly, so that you can use it more easily.



Get-EventLog help

Use the help for **Get-EventLog** as an example. If you type the command **Get-Help Get-EventLog** in the console and press Enter, the help returns the following syntax:

```
Get-EventLog [-LogName] <String> [<-InstanceId> <Int64[]>] [-After <DateTime>] [-AsBaseObject] [-Before <DateTime>] [-ComputerName <String[]>] [-EntryType {Error | Information | FailureAudit | SuccessAudit | Warning}] [-Index <Int32[]>] [-Message <String>] [-Newest <Int32>] [-Source <String[]>] [-UserName <String[]>] [<CommonParameters>]
Get-EventLog [-AsString] [-ComputerName <String[]>] [-List] [<CommonParameters>]
```

The two blocks of text are *parameter sets*, each of which represents one way in which you can run the command. Notice that each parameter set has many parameters, and they both have several parameters in common. You cannot mix and match parameters between sets. That is, if you decide to use the **-List** parameter, you cannot also use **-LogName**, because those two do not appear together in the same parameter set.

In the first parameter set, the **-LogName** parameter is mandatory. You can determine this because the entire parameter, which includes the name and the value, is *not* enclosed in square brackets. The help syntax also shows that the parameter accepts **<string>** values, meaning strings of letters, numbers, and other characters.

The **-LogName** parameter name is enclosed in square brackets, meaning it is a *positional parameter*. You cannot run the command without a log name. However, you do not have to type the **-LogName** parameter name. You do need to pass the log name string as the first parameter, because that is the position in the help file where the **-LogName** parameter appears. Therefore, the following two commands provide the same results:

```
Get-EventLog -LogName Application
Get-EventLog Application
```

Omitting parameter names

Be cautious when omitting parameter names, for a few reasons. You cannot omit every parameter. The **-ComputerName** parameter, as one example, cannot have the parameter name omitted. In addition, you can quickly lose track of what goes where. When you provide parameter names, the parameters can appear in any order, as in the following command:

```
Get-EventLog -ComputerName LON-DC1 -LogName Application -Newest 10
```

However, when you omit a parameter name, you become responsible for putting everything in the correct order. The following command, for example, will not work because the log name is being passed in the wrong position:

```
Get-EventLog -ComputerName LON-DC1 Application
```

Specifying multiple values

Some parameters accept more than one value. In the **SYNTAX** section, a double-square-bracket notation in the parameter value type designates these parameters. For example:

```
-ComputerName <string[]>
```

The above syntax indicates that the `-ComputerName` parameter can accept one or more string values. One easy way to specify multiple values is in a comma-separated list. You do not have to enclose the values in quotation marks, unless the values themselves contain a comma, or if they contain white space, such as a space or tab character. For example, use the following command to specify multiple computer names:

```
Get-EventLog -LogName Application -ComputerName LON-CL1,LON-DC1
```

 **Note:** You can find more information about each parameter by viewing the command's full help. For example, run **Get-Help Get-EventLog -Full** to see the full help for **Get-EventLog**, and notice the additional information displayed. You can, for example, confirm that the `-LogName` parameter is mandatory and appears in the first position.

 **Best Practice:** If you are just getting started with Windows PowerShell, try to provide full parameter names instead of passing parameter values by position. Full parameter names make commands easier to read and troubleshoot, and they make it easier to see when you are typing the command incorrectly.

Updating help

Windows PowerShell 3.0 and newer versions do not ship with help files. Instead, help files are available as an online service. Microsoft-authored commands have their help files hosted on a Microsoft-owned web server. Non-Microsoft commands can also use downloadable help, if the author or vendor builds the module correctly and provides an online location for the help files. Therefore, authors who write commands, including Microsoft authors, can make corrections and improvements to their help files over time, and deliver them without having to create an entire product update.

- Windows PowerShell 3.0 and newer versions do not ship with help files
- **Update-Help:**
 - Uses downloadable help content to update your local help
 - Checks no more than once every 24 hours by default
 - **Save-Help** enables you to download help and save it to an alternate location accessible to computers that are not connected to the Internet

Run **Update-Help** to scan your computer for all installed modules, retrieve online help locations from each, and try to download help files for each. You must run this command as a member of the local **Administrators** group, because Windows PowerShell core command help is stored in the `%systemdir%` folder. Be aware that error messages will display if help is not downloadable. In such cases, Windows PowerShell will still create a default help display for the commands.

Windows PowerShell defaults to downloading help files in your system's configured language. If help is not available in that language, Windows PowerShell defaults to the **en-US** (US English) language. You can override this behavior by using a parameter of **Update-Help** to specify the *UICulture* for which you want to retrieve help.

By default, **Update-Help** will check for help files only once every 24 hours, even if you run the command multiple times in a row. To override this behavior, specify the command's **-Force** parameter.

 **Note:** Your lab virtual machines might not be connected to the Internet and thus might be unable to run **Update-Help** successfully. For your convenience, the virtual machines were populated with the latest help content when they were created.

The companion to **Update-Help** is **Save-Help**. It downloads the help content but saves it to a location that you specify. You can then access that content by using it directly, or by physically moving the content to computers that are not connected to the Internet. **Update-Help** offers a parameter to specify an alternative source location. This enables those disconnected computers to update from that source location.

Prior to Windows PowerShell 4.0, **Update-Help** and **Save-Help** download help only for cmdlets that are installed on the computer where you run the command. They will not download help for cmdlets on other computers. In Windows PowerShell 4.0 and newer, you can use **Save-Help** for modules installed on remote computers.

About files

Although much of the help content in Windows PowerShell relates to commands, there are also many help files that describe Windows PowerShell concepts. These files include information about the Windows PowerShell scripting language, operators, and other details. This information does not specifically relate to a single command, but to global shell techniques and features.

You can see a complete list of these topics by running **Get-Help about***, and then viewing a single topic by running **Get-Help topicname**, such as **Get-Help about_common_parameters**.

These commands do not use the **-Example** or **-Full** parameters of the **Help** command. However, they are compatible with the **-ShowWindow** and **-Online** parameters.

When you use wildcard characters with the **Get-Help** command, About help files will appear in a list when their titles contain a match for your wildcard pattern. Typically, About help files will appear last, after any commands whose names also matched your wildcard pattern. You can also use the **-Category** parameter to specify a search for About files.

- Provide documentation for global shell techniques, concepts, and features
- Start with **about_**
- View list by running **Get-Help about***
- You will need to read many of these files to complete several upcoming lab exercises

 **Note:** For much of the rest of this course, you will refer to these About files for additional documentation. Frequently, you must read these files to discover the steps and techniques you need to complete lab exercises.

Demonstration: Using About files

In this demonstration, you will see how to use the About help file topics.

Demonstration Steps

1. View a list of About help file topics.
2. View the **about_aliases** help topic.
3. View the **about_eventlogs** help topic in a window.
4. View a help topic that will explain how to make the console beep.

Question: How would you search for a cmdlet that retrieves the properties of a computer from Active Directory?

Check Your Knowledge

Question	
You wish to join multiple computers to the Adatum domain. The Add-Computer cmdlet's <i>-ComputerName</i> parameter accepts multiple values. Which of the following is a set of valid values for this parameter?	
Select the correct answer.	
	-ComputerName LON-CL2;LON-CL3;LON-CL4
	-ComputerName "LON-CL2, LON-CL3, LON-CL4"
	-ComputerName LON-CL2 LON-CL3 LON-CL4
	-ComputerName LON-CL2, -ComputerName LON-CL3 -ComputerName LON-CL4
	-ComputerName LON-CL2,LON-CL3,LON-CL4

Lesson 3

Finding commands

In this lesson, you will learn how to find Windows PowerShell cmdlets for performing specific tasks. There are far too many cmdlets for you to memorize all their names. This lesson provides strategies and tools for finding cmdlets based on the actions they perform and the features or technologies they manage. Throughout this course, you will use these strategies to identify the cmdlets that you need to complete tasks and to learn how to use those cmdlets.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the relationship between modules and cmdlets.
- View modules.
- Identify options for finding cmdlets.
- Search for cmdlets.
- Describe aliases.
- Use aliases.
- Learn command syntax.
- Explain how to use **Show-Command**.

What are modules?

Modules are groups of related Windows PowerShell functionalities that are packaged together as a unit. For the purposes of this class, you can view modules simply as containers for cmdlets. Modules help to organize cmdlets and make them more distributable. Microsoft and other software companies provide modules as part of the management tools for their applications and services.

To use the cmdlets contained in a module, the module must be loaded into the current Windows PowerShell session. Some server products such as Exchange Server provide a shortcut to what appears to be a dedicated management shell. However, this is really a normal Windows PowerShell console session with modules specific to the application already loaded.

- **Modules:**
 - Are containers for related cmdlets
 - Are provided as part of management tools for various software packages
 - Must be loaded into current session
- Windows PowerShell version 3.0 and newer support autoloading
- Autoloading requires Windows Server 2012/Windows 8 or newer

Autoloading

In Windows PowerShell version 3.0 and newer, modules *autoload* if you run a cmdlet that is not currently loaded. This works if the module that contains the cmdlet is in a folder under the module load paths. By default, these are **%systemdir%\WindowsPowerShell\v1.0\Modules** and **%userprofiles%\Documents\WindowsPowerShell\Modules**. Within Windows PowerShell, the **Get-Help** command uses autoloading when searching for help topics. The **Get-Command** command also uses autoloading.

 **Note:** Within Windows PowerShell, the path `%systemdir%\WindowsPowerShell\v1.0\Modules` is commonly referred to by using the `$PSHome` environment variable and the path `$PSHome\Modules`.

 **Note:** An upcoming topic, "Searching for cmdlets," discusses autoloading in more detail.

Operating system compatibility

Keep in mind that some modules and cmdlets will only work with certain operating systems. For example, you can only import the Active Directory module on systems that are running Windows Server 2008 R2 or newer or have the Remote Server Administration Tools (RSAT) installed.

Demonstration: Viewing modules

In this demonstration, you will see how to find installed modules and also see how autoloading and manual loading of modules work.

Demonstration Steps

1. Open the **Windows PowerShell** console as an administrator on **LON-DC1**.
2. Display a list of currently loaded modules.
3. Run the cmdlet that returns a list of Active Directory users.
4. Display the updated list of currently loaded modules.
5. Display a list of the currently available modules, including those that are not loaded.
6. Import the module that contains cmdlets for managing features installed on a server and display the updated list of loaded modules.

Finding cmdlets

Windows PowerShell has extensive built-in help that typically includes examples. This makes it easier for you to learn how to use a cmdlet.

Finding the cmdlet that you need is often the challenge. For example, what cmdlet would you use to set an IP address on a network adapter or to disable a user account in Active Directory?

You can start by using what you know about the structure of cmdlet names, along with the **Get-Command** command or the **Get-Help** command.

Get-Command retrieves information about a command, or several commands, such as the name, category, version, and even the module that contains it. **Get-Help** retrieves help content about the command.

- Use **Get-Command** and **Get-Help**, both of which support wildcards
- Use **-Noun**, **-Verb**, and **-Module** parameters with **Get-Command**
- **Get-Help** can also search the content of help files if no match is found when searching command names
- Use the **PowerShellGet** module to find modules and commands from the PowerShell Gallery

Like the **Get-Help** command, the **Get-Command** accepts wildcard characters. This means that you can run the **Get-Command *event*** command and retrieve a list of commands that contain the text **event** in the name. **Get-Command** also has several parameters that you can use to further filter the results returned. For example, you can use the **-Noun** and **-Verb** parameters to filter out the noun and verb portions of the name, respectively. Both parameters accept wildcards, though in most cases you will not need to use wildcards with verbs. You can even combine the parameters to further refine the results returned. Run the **Get-Command -Noun event* -Verb Get** command to see a list of commands that have nouns starting with **event** and that use the **Get** verb.

When you make a guess about command names, try to use just the noun portion, and consider just a single-word, singular noun. For example, *event* and *log* might be good guesses when you are trying to find a command that works with Windows event logs.

Using modules to discover cmdlets

In the “Viewing modules” demonstration, you saw that when you use the **Get-Module** command, a partial list of cmdlets contained in a module display. However, you can use the module in another way to find cmdlets.

For example, if you have discovered the module **NetAdapter**, you would expect that it should have cmdlets used for managing network adapters. You can find all the commands contained in that module by running the **Get-Command -Module NetAdapter** command. The **-Module** parameter restricts the results to just those commands in the designated module.

Using Get-Help to discover cmdlets

As you learned earlier, you can perform similar searches by using **Get-Help**, including using wildcards. One advantage of using **Get-Help** instead of **Get-Command** is that **Get-Help** will perform a full-text search by using your query string if it cannot find a command name that matches. If you run the **Get-Command *beep*** command, no results are available. If you run the **Get-Help *beep*** command, multiple results are available.

You can also refer to the **Related Links** section of a cmdlet that you know is related to the one you are looking for. This section of the help topic lists related cmdlets, among other things.

Finding cmdlets on the Internet

You are not limited to searching for cmdlets that your computer already has installed. You can find a wide variety of Microsoft and non-Microsoft modules and cmdlets by searching the Internet. If you simply search by using the terms **PowerShell** and the technology you are working with, you will find a wide variety of links to articles on TechNet, Microsoft Developer Network (MSDN), and non-Microsoft websites. Virtually all Microsoft teams create cmdlets for use in managing their products, and you can install them as part of their management tools.

PowerShell Gallery

The PowerShell Gallery is a central repository for Windows PowerShell-related content, including scripts and modules. The PowerShell Gallery uses the Windows PowerShell module, **PowerShellGet**. This module is part of Windows PowerShell 5.0 and newer.

 **Additional Reading:** For more information on Windows PowerShell, refer to: “PowerShell Gallery” at: <https://aka.ms/last9g>

PowerShellGet contains cmdlets for finding and installing modules, scripts, and commands from the online gallery. For example, the **Find-Command** cmdlet searches for commands, functions, and aliases. It works very much like the **Get-Command** cmdlet, including support for wildcards.

You can pass the results of the **Find-Command** cmdlet to the **Install-Module** cmdlet, which the **PowerShellGet** module also contains. **Install-Module** will install the module that contains the cmdlet that you discovered.

 **Note:** You can also use the Command Add-On pane in the ISE application to find commands. It lists all the installed commands alphabetically, and lets you type a partial command name to see matching commands.

Demonstration: Searching for cmdlets

In this demonstration, you will see how to use several techniques to discover new cmdlets.

Demonstration Steps

1. Show a list of commands that deal with IPv4 addresses.
2. There is a command able to read Windows Event Logs (actually, there are two). Find one that can read event logs from a remote computer in addition to the local one.

What are aliases?

If you have experience using the **cmd.exe** CLI, you are familiar with batch commands such as:

- **dir**, for listing files and folders.
- **cd**, for changing directories.
- **mkdir**, for creating new directories.

In many cases, you can continue to use these commands within Windows PowerShell. However, behind the scenes, these commands are running native cmdlets. The **dir** command runs **Get-ChildItem**, the **cd** command runs **Set-Location**, and the **mkdir** command runs **New-Item**. These alternative commands work because they are *aliases* for the cmdlets that perform the action.

- Familiar batch commands include:
 - **Dir**
 - **Cd**
 - **Mkdir**
 - **Type**
- These are really aliases to Windows PowerShell commands
- External commands such as **ping.exe** and **ipconfig.exe** all work as usual
- Windows PowerShell commands often have a different syntax, even if accessed by an alias that matches an older command name

Aliases and parameters

It is important to note that the aliases typically do not use the parameters that the original commands used. For example, if you run the command **dir /s** in the console, you will receive an error, because **Get-ChildItem** does not understand the **/s** parameter. Instead, you must use the **-Recurse** parameter (**dir -Recurse**) to list the contents of the current folder and all its subfolders.

Get-Alias

Windows PowerShell includes more than just aliases for old batch and Linux commands. It also provides other aliases, such as **gci** for **Get-ChildItem** that you can use to enter commands quickly. You can discover aliases, their definitions, and the commands that they run, by using the **Get-Alias** cmdlet.

Get-Alias with no parameters will return all aliases defined. You can use the **-Name** parameter, a positional parameter, which also accepts wildcards, to find the definition for specific aliases. For example, running the command **Get-Alias di*** will return the aliases for both **diff** and **dir**.

You can also use the **Get-Alias** cmdlet to discover new cmdlets. For example, you used the batch command **del** to delete a file or folder. You can type the command **Get-Alias del** to discover that **del** is an alias for **Remove-Item**. You can even reverse the discovery process by running the command **Get-Alias -definition Remove-Item** to discover that **Remove-Item** has several other aliases, including **rd**, **erase**, and **ri**.

Parameters can also have aliases. For example, the **-s** parameter is an alias for **-Recurse** in the **Get-ChildItem** cmdlet. In fact, for parameters, you can use partial parameter names just like aliases, if the portion of the name you do include in the command is enough to uniquely identify that parameter.

New-Alias

You can also create your own alias by using the **New-Alias** cmdlet. This allows you to define your own custom alias that you can map to any existing cmdlet. Keep in mind, however, that custom aliases are not saved between Windows PowerShell sessions. You will need to use a Windows PowerShell profile to recreate the alias every time you open Windows PowerShell. Module 12, "Using advanced Windows PowerShell techniques" covers the creation of profiles.

Disadvantages of aliases

Aliases can help you type commands more quickly, but they tend to make scripts harder to read and understand. One reason is that the verb-noun syntax clearly defines the action taking place. It creates commands that read and sound more like natural language. Aliases for parameters and partial parameter names make scripts even harder to read. In most cases, using tab completion will make command entry almost as fast as entering an alias name, and will be far more accurate.

Demonstration: Using aliases

In this demonstration, you will see how to:

- Find an alias for a cmdlet.
- Find a cmdlet based on an alias you already know.
- Create an alias.

Demonstration Steps

1. Run the **dir** and **Get-ChildItem** commands, and then compare the results.
2. View the definition for the **dir** alias.
3. Create a new alias, **list**, for the **Get-ChildItem** command.
4. Run the **list** command and compare the results to those of **dir** and **Get-ChildItem**.
5. Show the definition for the **list** alias.
6. Show the various aliases for **Get-ChildItem**.

Using Show-Command

The **Show-Command** cmdlet opens a window that displays either a list of commands or the parameters for a specific command. This is the same window that displays when you select the **Show Command Window** option in the ISE.

To display the parameters for a specific command, provide the name of the command as the value for the **-Name** parameter. For example, to open the **Show Command Window** with the command used to retrieve an Active Directory user, type the following command in the console, and then press Enter:

```
Show-Command -Name Get-ADUser
```

The **-Name** parameter is positional, so the following will produce the same result:

```
Show-Command Get-ADUser
```

If you select the **Show Command Window** option in the ISE, and your cursor is within or immediately next to a command name within the console or scripting pane, the results will be the same.

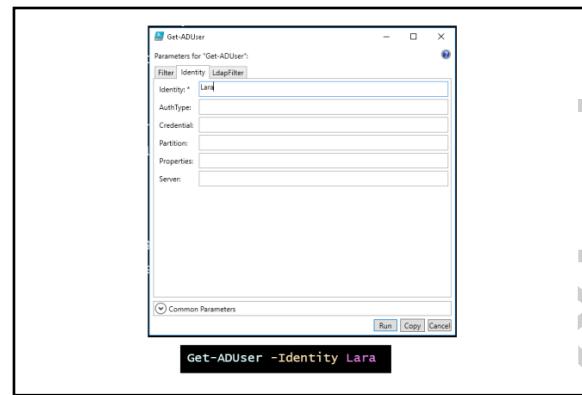
 **Note:** In these examples, **Show-Command** is the command that you are actually running, but **Get-WinEvent** is the name of the command that you want to see in the dialog box.

Within the **Show Command Window**, each parameter set for the specified command displays on a separate tab. This makes it visually clear that you cannot mix and match parameters between sets.

Once you have provided values for all the required parameters, you can run the command immediately by selecting **Run** in the **Show** commands window. You can also copy it to the Clipboard by selecting **Copy**. From the Clipboard, you can paste the command into the console, so that you can see the correct command-line syntax, without running the command.

Notice that **Show-Command** also exposes the Windows PowerShell *common parameters*, which are a set of parameters that Windows PowerShell adds to all commands to provide a specific set of consistent, baseline functionalities. You will learn more about many of the common parameters in upcoming modules. However, if you want to read about them immediately, you can run **help about_common_parameters** in Windows PowerShell.

Question: What is the difference between **Get-Help** and **Get-Command**? Why might they return different results for the same query?



Studijní materiály pro skolení

Lab B: Finding and running basic commands

Scenario

You are preparing to complete several administrative tasks by using Windows PowerShell. You need to discover commands that you will use to perform those tasks, run several commands to begin performing those tasks, and learn about new Windows PowerShell features that will enable you to complete those tasks.

Objectives

After completing this lab, you will be able to:

- Find new Windows PowerShell commands.
- Run basic Windows PowerShell commands.
- Use Windows PowerShell About topics to learn new shell concepts and techniques.

Lab Setup

Estimated Time: **45 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **ADATUM\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Finding commands

Scenario

In this exercise, you will use Windows PowerShell's **Get-Help** and **Get-Command** commands to discover new commands that can complete specific tasks within Windows PowerShell. In your tasks, italicized terms represent keyword clues to help you complete the task.

The main task for this exercise is as follows:

1. Find commands that will accomplish specified tasks.

► **Task 1: Find commands that will accomplish specified tasks**

- On **LON-CL1**, ensure that you are signed in as **Adatum\Administrator** and determine answers to the following questions:
 - What command would you run to resolve a DNS name?
 - What command would you run to make changes to a network adapter? After finding such a command, what parameter would you use to change its MAC address (on adapters that support changes to their MAC address)?
 - What command would let you enable a previously disabled scheduled task?
 - What command would let you block access to a file share by a particular user?
 - What command would you run to clear your computer's local **BranchCache** cache?
 - What command would you run to display a list of Windows Firewall rules? What parameter of that command would display only enabled rules?
 - What command would you run to display a list of all locally bound IP addresses?
 - What command would you run to suspend an active print job in a print queue?
 - What native Windows PowerShell command would you run to read the content of a text file?

Results: After completing this exercise, you will have demonstrated your ability to find new Windows PowerShell commands that perform specific tasks.

Exercise 2: Running commands

Scenario

In this exercise, you will run several basic Windows PowerShell commands. In some instances, you might have to find the commands that you will use to complete the task.

The main task for this exercise is as follows:

1. Run commands to accomplish specified tasks.

► **Task 1: Run commands to accomplish specified tasks**

1. Ensure you are signed in on the **LON-CL1** virtual machine as **Adatum\Administrator**.
2. Display a list of enabled Windows Firewall rules.
3. Display a list of all local IPv4 addresses.
4. Set the startup type of the BITS service to **Automatic**:
 - a. Open the **Computer Management** console and go to **Services and Applications**.
 - b. Locate the Background Intelligence Transfer Service (BITS) and note its startup type setting prior to and after changing the startup type in Windows PowerShell.

5. Test the network connection to **LON-DC1**. Your command should return only a True or False value, without any other output.
6. Display the newest 10 entries from the local Security event log.

Results: After completing this exercise, you will have demonstrated your ability to run Windows PowerShell commands by using correct command-line syntax.

Exercise 3: Using About files

Scenario

In this exercise, you will use help discovery techniques to find content in About files, and then use that content to answer questions about global Windows PowerShell functionality.

Words in italic are clues. Remember that you must use **Get-Help** and wildcard characters. Because About files are not commands, **Get-Command** will not be useful in this exercise.

The main tasks for this exercise are as follows:

1. Locate and read About help files.
2. Prepare for the next module.

► Task 1: Locate and read About help files

- Ensure that you are still signed in to **LON-CL1** as **Adatum\Administrator** from the previous exercise, and answer the following questions:
 - What comparison operator does Windows PowerShell use for wildcard string comparisons?
 - Are Windows PowerShell comparison operators typically case-sensitive?
 - How would you use **\$Env** to display the **COMPUTERNAME** environment variable?
 - What external command could you use to create a self-signed digital certificate that is usable for signing Windows PowerShell scripts?

Results: After completing this exercise, you will have demonstrated your ability to locate help content in About files.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Question: What are some methods for finding commands, other than using **Get-Help** and **Get-Command**?

Module Review and Takeaways

Best Practice

When you discover a new command, either by using **Get-Help** or **Get-Command**, or by reading about the command somewhere, always read the command's help file and learn about its additional capabilities.

Even familiar commands can gain new functionality in new versions of Windows PowerShell. Read the help files even of commands that you already know well from earlier versions, to see what new features might exist.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Help files contain only syntax section—no description or examples.	
Cannot use Update-Help with computers not connected to the Internet.	
Update-Help did not download all help.	

Review Question

Question: What functionality does the ISE in Windows PowerShell 5.0 now have that was previously only available in the console?

Studijní materiály Okškolení

Module 2

Cmdlets for administration

Contents:

Module Overview	2-1
Lesson 1: Active Directory administration cmdlets	2-2
Lesson 2: Network configuration cmdlets	2-13
Lesson 3: Other server administration cmdlets	2-19
Lab: Windows administration	2-24
Module Review and Takeaways	2-28

Module Overview

Each time you need to accomplish a task in the Windows PowerShell command-line interface, you could search for the cmdlets you need. However, you will be a more productive administrator if you have at least a basic understanding of the cmdlets that are available for system administration so that you do not need to search for them at all. Even if you do need to search for cmdlets, you can predict naming patterns and find them more quickly. In this module, you will learn about the cmdlets that you will commonly use for administration.

Objectives

After completing this module, students will be able to:

- Identify and use cmdlets for Active Directory administration.
- Identify and use cmdlets for network configuration.
- Identify and use cmdlets for other server administration tasks.

Lesson 1

Active Directory administration cmdlets

Active Directory Domain Services (AD DS) and its related service form the core of Windows Server–based networks. The AD DS database stores information about all the objects that make up the network, such as accounts for users, computers, and groups. The AD DS database is searchable and provides a mechanism for applying configuration and security settings for all of those objects.

The Active Directory module for Windows PowerShell gives you the ability to automate AD DS administration. The use of Windows PowerShell for AD DS administration speeds up administration by allowing you to make bulk updates instead of updating AD DS objects individually. In this lesson, you will learn about the cmdlets for administering AD DS. To find Active Directory cmdlets, look for the prefix "AD," which most Active Directory cmdlets have in the noun part of the cmdlet name.

Lesson Objectives

After completing this lesson, students will be able to:

- Identify user management cmdlets.
- List group management cmdlets.
- Manage users and groups.
- Describe the cmdlets for managing computer objects.
- Describe the cmdlets for managing organizational units (OUs).
- Describe the cmdlets for managing Active Directory objects.
- Manage Active Directory objects.

User management cmdlets

User management is a core responsibility of administrators. Windows PowerShell cmdlets allow you to create, modify, and delete user accounts individually or in bulk. User account cmdlets have the text "User" or "Account" in the noun part of the name. Include one or the other in wildcard name searches when you are using **Get-Help** or **Get-Command**.

Cmdlet	Description
New-ADUser	Creates a user account
Set-ADUser	Modifies properties of a user account
Remove-ADUser	Deletes a user account
Set-ADAccountPassword	Resets the password of a user account
Set-ADAccountExpiration	Modifies the expiration date of a user account
Unlock-ADAccount	Unlocks a user account after it has become locked after too many incorrect sign-in attempts
Enable-ADAccount	Enables a user account
Disable-ADAccount	Disables a user account

New-ADUser "Jane Doe" -Department IT

The following table lists some common cmdlets for managing user accounts.

Cmdlet	Description
New-ADUser	Creates a user account
Get-ADUser	Retrieves a user account
Set-ADUser	Modifies properties of a user account
Remove-ADUser	Deletes a user account
Set-ADAccountPassword	Resets the password of a user account
Set-ADAccountExpiration	Modifies the expiration date of a user account
Unlock-ADAccount	Unlocks a user account that has been locked after exceeding the accepted number of incorrect sign-in attempts
Enable-ADAccount	Enables a user account
Disable-ADAccount	Disables a user account

Retrieving users

The **Get-ADUser** cmdlet requires that you identify the user or users that you want to retrieve. You can do this by using the *-Identity* parameter, which accepts one of several property values, including the Security Accounts Manager (SAM) account name or distinguished name.

Windows PowerShell only returns a default set of properties when you use **Get-ADUser**. To view other properties, you will need to use the *-Properties* parameter with a comma-separated list of properties or the "*" wildcard.

For example, you can retrieve the default set of properties along with the department and email address of a user with the SAM account **janedoe** by typing the following command in the console, and then pressing Enter:

```
Get-ADUser -Identity janedoe -Properties Department,EmailAddress
```

The other way to specify a user or users is with the *-Filter* parameter. The *-Filter* parameter accepts a query based on regular expressions, which later modules in this course cover in more detail. For example, to retrieve all AD DS users and their properties, type the following command in the console, and then press Enter:

```
Get-ADUser -Filter * -Properties *
```

 **Note:** The help for **Get-ADUser** includes examples that use features of Windows PowerShell that you will learn about later in this course. For example, more advanced *-Filter* operations require comparison operators, which you will learn about in the next module.

Creating new user accounts

When you use the **New-ADUser** cmdlet to create new user accounts, the *-Name* parameter is required.

You can also set most other user properties, including a password. When you create a new account, consider the following points:

- If you do not use the *-AccountPassword* parameter, then no password is set and the user account is disabled. You cannot set the *-Enabled* parameter as **\$true** when no password is set.
- If you use the *-AccountPassword* parameter to specify a password, then you must specify a variable that contains the password as a secure string or choose to enter the password from the console. A secure string is encrypted in memory.
- If you set a password, then you can enable the user account by setting the *-Enabled* parameter as **\$true**.

The following table lists common parameters for the **New-ADUser** cmdlet.

Parameter	Description
<i>-AccountExpirationDate</i>	Defines the expiration date for a user account
<i>-AccountPassword</i>	Defines the password for a user account
<i>-ChangePasswordAtLogon</i>	Requires a user account to change passwords at the next sign-in
<i>-Department</i>	Defines the department for a user account
<i>-DisplayName</i>	Defines the display name for a user account
<i>-HomeDirectory</i>	Defines the location of the home directory for a user account
<i>-HomeDrive</i>	Defines the drive letters that map to the home directory for a user account
<i>-GivenName</i>	Defines the first name of a user account
<i>-Name</i>	Defines the name of a user account
<i>-Path</i>	Defines the OU or container where the user account is created
<i>-SamAccountName</i>	Defines the SAM account name for a user account
<i>-Surname</i>	Defines the last name of a user account

To add a user account in the IT department, type the following command in the console, and then press Enter:

```
New-ADUser "Jane Doe" -Department IT
```

Because no password was set, the account is not enabled, and you cannot enable it.



Note: Later in the course, you will learn how to read text files and comma-delimited files, which you can use for bulk creation of passwords.

Group management cmdlets

The management of Active Directory groups closely relates to the management of users. You can use Windows PowerShell cmdlets to create and delete groups and to modify group properties. You can also use these cmdlets to change the members who are assigned to a group. Additionally, you can use some cmdlets to modify the groups that are assigned to a user or another Active Directory object.

Managing groups

Cmdlets for modifying groups have the text "group" in their names. Those that modify group membership by adding members to a group, for example, have the text "groupmember" in their names. Cmdlets that modify the groups that a user, computer, or other Active Directory object is a member of have the text "principalgroupmembership" in their names.

The following table lists some cmdlets for managing groups.

Cmdlet	Description
New-ADGroup	Creates a new group
Set-ADGroup	Modifies properties of a group
Get-ADGroup	Displays properties of a group
Remove-ADGroup	Deletes a group
Add-ADGroupMember	Adds members to a group
Get-ADGroupMember	Displays membership of a group
Remove-ADGroupMember	Removes members from a group
Add-ADPrincipalGroupMembership	Adds group membership to an object
Get-ADPrincipalGroupMembership	Displays group membership of an object
Remove-ADPrincipalGroupMembership	Removes group membership from an object
New-ADGroup -Name "FileServerAdmins" -GroupScope Global	

Cmdlet	Description
New-ADGroup	Creates a new group
Set-ADGroup	Modifies properties of a group
Get-ADGroup	Displays properties of a group
Remove-ADGroup	Deletes a group
Add-ADGroupMember	Adds members to a group
Get-ADGroupMember	Displays members of a group
Remove-ADGroupMember	Removes members from a group
Add-ADPrincipalGroupMembership	Adds group membership to an object
Get-ADPrincipalGroupMembership	Displays group membership of an object
Remove-ADPrincipalGroupMembership	Removes group membership from an object

Creating new groups

You can use the **New-ADGroup** cmdlet to create groups. However, when you create groups by using the **New-ADGroup** cmdlet, you must use the *-GroupScope* parameter in addition to the group name. This is the only required parameter. The following table lists common parameters for **New-ADGroup**.

Parameter	Description
<i>-Name</i>	Defines the name of a group
<i>-GroupScope</i>	Defines the scope of a group as DomainLocal , Global , or Universal ; you must provide this parameter
<i>-DisplayName</i>	Defines the Lightweight Directory Access Protocol (LDAP) display name for an object
<i>-GroupCategory</i>	Defines whether a group is a security group or a distribution group; if you do not specify either, a security group is created
<i>-ManagedBy</i>	Defines a user or group that can manage a group
<i>-Path</i>	Defines the OU or container in which a group is created
<i>-SamAccountName</i>	Defines a name that is backward-compatible with older operating systems

To create a new group named **FileServerAdmins**, type the following command in the console, and then press Enter:

```
New-ADGroup -Name FileServerAdmins -GroupScope Global
```

Managing group membership

As mentioned earlier, you can use the ***-ADGroupMember** or the ***-ADPrincipalGroupMembership** cmdlets to manage group management in two different ways. The difference between the two is a matter of focusing on an object and modifying the groups to which it belongs, or focusing on the group and modifying the members that belong to it. Additionally, you can choose which set to use based on the decision to *pipe* a list of members to the command or provide a list of members.



Note: You will learn about piping in Module 4, "Understanding how the pipeline works."

- ***-ADGroupMember** cmdlets modify the membership of a group. For example, you can add or remove members of a group.
 - You can pass a list of groups to these cmdlets.
 - You cannot *pipe* a list of members to these cmdlets.
- ***-ADPrincipalGroupMembership** cmdlets modify the group membership of an object such as a user. For example, you can change a user account to add it as a member of a group.
 - You cannot provide a list of groups to these cmdlets.
 - You can pipe a list of members to these cmdlets.



Best Practice: Use `*-ADGroupMember` when speed is important because membership is modified as a single operation in AD DS. Loop operations, such as piping, change membership as a series of operations.

Demonstration: Managing users and groups

In this demonstration, you will see how to:

- Create a new global group in the IT department.
- Create a new user in the IT department.
- Add two users from the IT department to the **HelpDesk** group.
- Set the address for all **HelpDesk** group users.
- Verify the group membership for the new user.
- Verify the updated user properties.

Demonstration Steps

Create a new global group in the IT department

1. On **LON-CL1**, start a Windows PowerShell session with elevated permissions.
2. Run the following command:

```
New-ADGroup -Name HelpDesk -Path "ou=IT,dc=Adatum,dc=com" -GroupScope Global
```

Create a new user in the IT department

- Run the following command:

```
New-ADUser -Name "Jane Doe" -Department "IT"
```

Add two users from the IT department to the HelpDesk group

- Run the following command:

```
Add-ADGroupMember "HelpDesk" -Members "Lara","Jane Doe"
```

Set the address for a HelpDesk group user

1. Run the following command:

```
Get-ADGroupMember HelpDesk
```

2. Run the following command:

```
Set-ADUser Lara -StreetAddress "1530 Nowhere Ave." -City "Winnipeg" -State "Manitoba" -Country "CA"
```

Verify the group membership for the new user

- Run the following command:

```
Get-ADPrincipalGroupMembership "Jane Doe"
```

Verify the updated user properties

- Run the following command:

```
Get-ADUser Lara -Properties StreetAddress,City,State,Country
```

Computer object management cmdlets

The Active Directory module also has cmdlets to create, modify, and delete computer accounts. You can use these cmdlets for individual operations or as part of a script to perform bulk operations. The cmdlets for managing computer objects have the text "computer" in their names.

The following table lists cmdlets that you can use to manage computer accounts.

Cmdlet	Description
New-ADComputer	Creates a new computer account
Set-ADComputer	Modifies properties of a computer account
Get-ADComputer	Displays properties of a computer account
Remove-ADComputer	Deletes a computer account
Test-ComputerSecureChannel	Verifies or repairs the trust relationship between a computer and a domain
Reset-ComputerMachinePassword	Resets the password for a computer account

New-ADComputer -Name LON-CL10 -Path "ou=marketing,dc=adatum,dc=com" -Enabled \$true

Cmdlet	Description
New-ADComputer	Creates a new computer account
Set-ADComputer	Modifies properties of a computer account
Get-ADComputer	Displays properties of a computer account
Remove-ADComputer	Deletes a computer account
Test-ComputerSecureChannel	Verifies or repairs the trust relationship between a computer and the domain
Reset-ComputerMachinePassword	Resets the password for a computer account

Creating new computer accounts

You can use the **New-ADComputer** cmdlet to create a new computer account before you join the computer to the domain. You do this so that you can create the computer account in the correct OU before deploying the computer.

The following table lists common parameters for **New-ADComputer**.

Parameter	Description
-Name	Defines the name of a computer account
-Path	Defines the OU or container where a computer account is created
-Enabled	Defines whether the computer account is enabled or disabled; by default, a computer account is enabled and a random password is generated

The following is an example of a command that you can use to create a computer account:

```
New-ADComputer -Name LON-CL10 -Path "ou=marketing,dc=adatum,dc=com" -Enabled $true
```

Repairing the trust relationship for a computer account

You can use the **Test-ComputerSecureChannel** cmdlet with the **-Repair** parameter to repair a lost trust relationship between a computer and a domain. You must run the cmdlet on the computer with the lost trust relationship.

Account vs. device management cmdlets

*-**ADComputer** cmdlets are part of the Active Directory module and manage the computer object, not the computer. *-**Computer** cmdlets manage the physical device. For example, you can use the **Add-Computer** cmdlet to join a computer to a domain.

OU management cmdlets

Windows PowerShell provides cmdlets that you can use to create, modify, and delete OUs. Like the cmdlets for users, groups, and computers, you can use these cmdlets for individual operations or as part of a script to perform bulk operations.

OU management cmdlets have the text “organizationalunit” in the name. The following table lists the cmdlets that you can use to manage OUs.

Cmdlet	Description
New-ADOrganizationalUnit	Creates an OU
Set-ADOrganizationalUnit	Modifies properties of an OU
Get-ADOrganizationalUnit	Displays properties of an OU
Remove-ADOrganizationalUnit	Deletes an OU

New-ADOrganizationalUnit -Name Sales -Path "ou=marketing,dc=adatum,dc=com" -ProtectedFromAccidentalDeletion \$true

Cmdlet	Description
New-ADOrganizationalUnit	Creates an OU
Set-ADOrganizationalUnit	Modifies properties of an OU
Get-ADOrganizationalUnit	Displays properties of an OU
Remove-ADOrganizationalUnit	Deletes an OU

Creating new OUs

You can use the **New-ADOrganizationalUnit** cmdlet to create a new OU to represent departments or physical locations within your organization. The following table shows common parameters for the **New-ADOrganizationalUnit** cmdlet.

Parameters	Description
-Name	Defines the name of a new OU
-Path	Defines the location of a new OU
-ProtectedFromAccidentalDeletion	Prevents anyone from accidentally deleting an OU; the default value is \$true

The following is an example of a command to create a new OU:

```
New-ADOrganizationalUnit -Name Sales -Path "ou=marketing,dc=adatum,dc=com" -  
ProtectedFromAccidentalDeletion $true
```

Active Directory object cmdlets

You will sometimes need to manage Active Directory objects that do not have their own management cmdlets, such as contacts. You might also want to manage multiple object types in a single operation, such as moving users and computers from one OU to another OU. The Active Directory module provides cmdlets that allow you to create, delete, and modify these objects and their properties. Because these cmdlets can manage all objects, they repeat some functionality of the cmdlets for managing users, computers, groups, and OUs.

Cmdlet	Description
New-ADObject	Creates a new Active Directory object
Set-ADObject	Modifies properties of an Active Directory object
Get-ADObject	Displays properties of an Active Directory object
Remove-ADObject	Deletes an Active Directory object
Rename-ADObject	Renames an Active Directory object
Restore-ADObject	Restores a deleted Active Directory object from the Active Directory recycle bin
Move-ADObject	Moves an Active Directory object from one container to another container
Sync-ADObject	Syncs an Active Directory object between two domain controllers

New-ADObject -Name "JohnSmithcontact" -Type contact

***-ADObject** cmdlets sometimes perform faster than cmdlets that are specific to object type. This is because those cmdlets add the cost of filtering the set of applicable objects to their operations.

Cmdlets for changing generic Active Directory objects have the text "Object" in the noun part of the name. The following table lists some cmdlets for managing Active Directory objects.

Cmdlet	Description
New-ADObject	Creates a new Active Directory object
Set-ADObject	Modifies properties of an Active Directory object
Get-ADObject	Displays properties of an Active Directory object
Remove-ADObject	Deletes an Active Directory object
Rename-ADObject	Renames an Active Directory object
Restore-ADObject	Restores a deleted Active Directory object from the Active Directory Recycle Bin
Move-ADObject	Moves an Active Directory object from one container to another container
Sync-ADObject	Syncs an Active Directory object between two domain controllers

Creating a new Active Directory object

You can use the **New-ADObject** cmdlet to create objects. When using **New-ADObject**, you must specify the name and the object type. The following table lists common parameters for **New-ADObject**.

Parameter	Description
-Name	Defines the name of an object
-Type	Defines the LDAP type of an object
-OtherAttributes	Defines properties of an object that is not accessible from other parameters
-Path	Defines the container in which an object is created

The following command creates a new contact object:

```
New-ADObject -Name "JohnSmithcontact" -Type contact
```

Demonstration: Managing Active Directory objects

In this demonstration, you will see how to:

- Create an Active Directory contact object that has no dedicated cmdlets.
- Verify the creation of the contact.
- Manage user properties by using Active Directory object cmdlets.
- Verify the property changes.
- Change the name of the **HelpDesk** group to **SupportTeam**.
- Verify the **HelpDesk** group name change.

Demonstration Steps

Create an Active Directory contact object that has no dedicated cmdlets

1. On **LON-CL1**, start a Windows PowerShell session with elevated permissions.
2. Run the following command:

```
New-ADObject -Name JohnSmithcontact -Type contact -DisplayName "John Smith  
(Contoso.com)"
```

Verify the creation of the contact

- Run the following command:

```
Get-ADObject -Filter 'ObjectClass -eq "contact"
```

Manage user properties by using Active Directory object cmdlets

- Run the following command:

```
Set-ADObject -Identity "CN=Lara Raisic,OU=IT,DC=Adatum,DC=com" -Description "Member  
of support team"
```

Verify the property changes

- Run the following command:

```
Get-ADUser Lara -Properties Description
```

Change the name of the HelpDesk group to SupportTeam

- Run the following command:

```
Rename-ADObject -Identity "CN=HelpDesk,OU=IT,DC=Adatum,DC=com" -NewName SupportTeam
```

Verify the HelpDesk group name change

- Run the following command:

```
Get-ADGroup HelpDesk
```



Note: Note that the **Name** and **DistinguishedName** properties changed, but not the **SAMAccountName** property.

Check Your Knowledge

Question	
Which of the following cmdlet verbs is not associated with the ADUser noun?	
Select the correct answer.	
	Get
	Update
	New
	Remove
	Set

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
The default value for the <i>-ProtectedFromAccidentalDeletion</i> parameter of New-ADOrganizationalUnit is \$true.	

Lesson 2

Network configuration cmdlets

Networking is another core functional area for administrators. In this lesson, you will learn about cmdlets for configuring networking in Windows Server. Windows PowerShell provides cmdlets for managing all aspects of Windows Server networks, including TCP/IP, Domain Name System (DNS), firewalls, and gateways. In addition to the cmdlets for managing network features and components, the **Test-NetConnection** cmdlet is also available. This cmdlet offers the same functionality as command-line interface tools such as **Ping.exe** and **Telnet.exe**, and you can use it to diagnose network configuration settings.

Lesson Objectives

After completing this lesson, students will be able to:

- Identify cmdlets for managing IP addresses.
- Describe how to change the default gateways.
- Describe how to modify DNS client configuration.
- List cmdlets for managing Windows Firewall.
- Configure network settings.

Managing IP addresses

Windows PowerShell offers a complete set of cmdlets for managing IP addresses on local and remote computers. You can use cmdlets to add, remove, and change IP addresses.

IP address management cmdlets use the noun "netipaddress" in their names. You can also find them by using the **Get-Command** command with the **-Module NetTCPIP** parameter.

The following table lists common cmdlets for managing IP addresses.

Cmdlet	Description
New-NetIPAddress	Creates a new IP address
Set-NetIPAddress	Sets properties of an IP address
Get-NetIPAddress	Displays properties of an IP address
Remove-NetIPAddress	Deletes an IP address

```
New-NetIPAddress -IPAddress 192.168.1.10 -InterfaceAlias "Ethernet" -PrefixLength 24 -DefaultGateway 192.168.1.1
```

Cmdlet	Description
New-NetIPAddress	Creates a new IP address
Get-NetIPAddress	Displays properties of an IP address
Set-NetIPAddress	Modifies properties of an IP address
Remove-NetIPAddress	Deletes an IP address

Creating new IP addresses

The **New-NetIPAddress** cmdlet requires an IPv4 or IPv6 address and either the alias or index of a network interface. As a best practice, you should also set the default gateway and subnet mask at the same time.

The following table lists common parameters for **Create-NetIPAddress**.

Parameter	Description
-IPAddress	Defines the IPv4 or IPv6 address to create
-InterfaceIndex	Defines the network interface, by index, for the IP address
-InterfaceAlias	Defines the network interface, by name, for the IP address
-DefaultGateway	Defines the IPv4 or IPv6 address of the default gateway host
-PrefixLength	Defines the subnet mask for the IP address

The following command creates a new IP address on the Ethernet interface:

```
New-NetIPAddress -IPAddress 192.168.1.10 -InterfaceAlias "Ethernet" -PrefixLength 24 -DefaultGateway 192.168.1.1
```

The **New-NetIPAddress** cmdlet also accepts the **-AddressFamily** parameter, which defines the IP address family. If you do not use this parameter, the address family property is detected automatically.



Note: Windows Server 2012 and newer also introduced IP Address Management (IPAM) server features, which include related cmdlets found in the IPAMServer module for Windows PowerShell. Those cmdlets have the text "IPAM" in the noun part of the name.

Managing routing

IP routing forwards data packets based on the destination IP address. This routing is based on routing tables, and while entries are made automatically, you might need to add, remove, or modify routing table entries manually. The cmdlets for managing routing table entries have the noun "NetRoute" in the names.

The following table lists common cmdlets for managing routing table entries.

Cmdlet	Description
New-NetRoute	Creates an IP routing table entry
Set-NetRoute	Sets properties of an IP routing table entry
Get-NetRoute	Displays properties of an IP routing table entry
Remove-NetRoute	Deletes an IP routing table entry
Find-NetRoute	Identifies the best local IP address and route to reach a remote address

```
New-NetRoute -DestinationPrefix 0.0.0.0/24 -InterfaceAlias "Ethernet" -DefaultGateway 192.168.1.1
```

Cmdlet	Description
New-NetRoute	Creates an entry in the IP routing table
Get-NetRoute	Retrieves an entry from the IP routing table
Set-NetRoute	Modifies properties of an entry in the IP routing table
Remove-NetRoute	Deletes an entry from the IP routing table
Find-NetRoute	Identifies the best local IP address and route to reach a remote address

Creating an IP routing table entry

You can use the **New-NetRoute** cmdlet to create routing table entries. The **New-NetRoute** cmdlet requires you to identify the network interface and destination prefix.

The following table lists common parameters for **New-NetRoute**.

Parameter	Description
<code>-DestinationPrefix</code>	Defines the destination prefix of an IP route
<code>-InterfaceAlias</code>	Defines the network interface, by alias, for an IP route
<code>-InterfaceIndex</code>	Defines the network interface, by index, for an IP route
<code>-NextHop</code>	Defines the next hop for an IP route
<code>-RouteMetric</code>	Defines the route metric for an IP route

The following command creates an IP routing table entry:

```
New-NetRoute -DestinationPrefix 0.0.0.0/24 -InterfaceAlias "Ethernet" -DefaultGateway  
192.168.1.1
```

Managing DNS clients

Managing DNS clients, including DNS servers and domain controllers, is essential to the health of your network. Windows PowerShell offers cmdlets for managing DNS client settings and the client resolve cache, and for securing DNS clients.

DNS client management cmdlets are part of the `DNSClient` module for Windows PowerShell and have the text "DnsClient" in the noun part of the name.

The following table lists common cmdlets for modifying client settings.

Cmdlet	Description
Get-DnsClient	Gets details about a network interface on a computer
Set-DnsClient	Set DNS client configuration settings for a network interface
Get-DnsClientServerAddress	Gets the DNS server address settings for a network interface
Set-DnsClientServerAddress	Sets the DNS server address for a network interface

```
Set-DnsClient -InterfaceAlias Ethernet -ConnectionSpecificSuffix  
"adatum.com"
```

Cmdlet	Description
Get-DnsClient	Gets details about a network interface
Set-DnsClient	Sets DNS client configuration settings for a network interface
Get-DnsClientServerAddress	Gets the DNS server address settings for a network interface
Set-DnsClientServerAddress	Sets the DNS server address for a network interface



Note: **Set-DnsClient** requires an interface that an alias or index references.

The following command sets the connection-specific suffix for an interface:

```
Set-DnsClient -InterfaceAlias Ethernet -ConnectionSpecificSuffix "adatum.com"
```

Managing Windows Firewall

Proper administration of Windows Firewall settings is essential to improving the security of networks and computers. You can use some Windows PowerShell cmdlets to manage firewall settings and rules.

You can find the cmdlets for managing Windows Firewall in the NetSecurity module for Windows PowerShell. The cmdlets have the text "NetFirewall" in their names, and the cmdlets for firewall rule management use the noun "NetFirewallRule."

The following table lists most of the cmdlets for managing firewall settings and rules.

Cmdlet	Description
New-NetFirewallRule	Creates a new firewall rule
Set-NetFirewallRule	Sets properties for firewall rules
Get-NetFirewallRule	Gets properties for firewall rules
Remove-NetFirewallRule	Deletes firewall rules
Rename-NetFirewallRule	Renames firewall rules
Copy-NetFirewallRule	Makes a copy of firewall rules
Enable-NetFirewallRule	Enables firewall rules
Disable-NetFirewallRule	Disables firewall rules
Get-NetFirewallProfile	Gets properties for firewall profiles
Set-NetFirewallProfile	Sets properties for firewall profiles

Cmdlet	Description
New-NetFirewallRule	Creates a new firewall rule
Set-NetFirewallRule	Sets properties for a firewall rule
Get-NetFirewallRule	Gets properties for a firewall rule
Remove-NetFirewallRule	Deletes a firewall rule
Rename-NetFirewallRule	Renames a firewall rule
Copy-NetFirewallRule	Makes a copy of a firewall rule
Enable-NetFirewallRule	Enables a firewall rule
Disable-NetFirewallRule	Disables a firewall rule
Get-NetFirewallProfile	Gets properties for a firewall profile
Set-NetFirewallProfile	Sets properties for a firewall profile

You can use the **Get-NetFirewallRule** cmdlet to retrieve settings for firewall rules. You can enable and disable rules by using one of the following:

- The **Set-NetFirewallRule** cmdlet with the *-Enabled* parameter
- The **Enable-NetFirewallRule** or **Disable-NetFirewallRule** cmdlets.

The following commands both enable firewall rules in the group **Remote Access**:

```
Enable-NetFirewallRule -DisplayGroup "Remote Access"
```

and

```
Set-NetFirewallRule -DisplayGroup "Remote Access" -Enabled True
```

Demonstration: Configuring network settings

In this demonstration, you will see how to:

- Test the network connection to **LON-DC1**.
- View the network configuration for **LON-CL1**.
- Change the client IP address.
- Change the DNS server for **LON-CL1**.
- Change the default gateway for **LON-CL1**.
- Confirm the network configuration changes.
- Test the effect of the changes.

Demonstration Steps

Test the network connection to LON-DC1

- Run the following command:

```
Test-Connection LON-DC1
```

 **Note:** Note the speed of the connection so that you can compare it to the speed after you make changes.

View the network configuration for LON-CL1

- Run the following command:

```
Get-NetIPConfiguration
```

 **Note:** Note the IP address, default gateway, and DNS server.

Change the client IP address

1. Run the following command:

```
New-NetIPAddress -InterfaceAlias Ethernet -IPAddress 172.16.0.30 -PrefixLength 16
```

2. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Remove-NetIPAddress -InterfaceAlias Ethernet -IPAddress 172.16.0.40
```

3. Type **Y** and press Enter twice to confirm the change.

Change the DNS server for LON-CL1

- Run the following command:

```
Set-DnsClientServerAddress -InterfaceAlias Ethernet -ServerAddress 172.16.0.11
```

Change the default gateway for LON-CL1

- Run the following command:

```
Remove-NetRoute -InterfaceAlias Ethernet -DestinationPrefix 0.0.0.0/0
```

- Type **Y** and press Enter twice to confirm the change.

- Run the following command:

```
New-NetRoute -InterfaceAlias Ethernet -DestinationPrefix 0.0.0.0/0 -NextHop  
172.16.0.2
```

Confirm the network configuration changes

- Run the following command:

```
Get-NetIPConfiguration
```

Test the effect of the changes

- Run the following command:

```
Test-Connection LON-DC1
```



Note: It now takes much longer to receive a response from **LON-DC1**.

Question: Which two parameters can you use with ***-NetIPAddress** cmdlets to identify a network interface?

Lesson 3

Other server administration cmdlets

Windows PowerShell offers many cmdlets that you can use to manage Windows features and services. Teaching you every available cmdlet is beyond the scope of this course; however, this course will familiarize you with the cmdlets that you are most likely to need in your work. In this lesson, you will learn about other cmdlets to administer Group Policy, **Server Manager**, Hyper-V, and Internet Information Services (IIS).

Lesson Objectives

After completing this lesson, students will be able to:

- Describe the cmdlets for managing Group Policy.
- Describe the cmdlets for managing server features, roles, and services.
- Describe the cmdlets for managing Hyper-V and virtual machines (VMs).
- Describe the cmdlets for managing IIS.

Group Policy management cmdlets

You can use Windows PowerShell to automate the management of most tasks involving Group Policy Objects (GPOs), including creating, deleting, backing up, reporting, and importing GPOs. You can also associate GPOs with AD DS containers, and you can set GPO inheritance and permissions on AD DS OUs.

Group Policy management cmdlets require Windows Server 2008 R2 or newer, or Windows 7 or newer with Remote Server Administration Tools installed. Group Policy management cmdlets are part of the **GroupPolicy** module for Windows

PowerShell. Cmdlet names include the prefix "GP" in the names, and most have "GPO" as the noun.

The following table lists some common GPO cmdlets.

Cmdlet	Description
New-GPO	Creates a new GPO
Get-GPO	Retrieves a GPO
Set-GPO	Modifies properties of a GPO
Remove-GPO	Deletes a GPO
Rename-GPO	Renames a GPO
Backup-GPO	Creates a backup of a GPO
Copy-GPO	Copies a GPO from one domain to another
Restore-GPO	Restores a GPO from backup files
New-GPLink	Links a GPO to an AD DS container
Import-GPO	Imports GPO settings from a backed-up GPO
Set-GPRegistryValue	Configures one or more registry-based policy settings in a GPO

Cmdlet	Description
New-GPO	Creates a new GPO
Get-GPO	Retrieves a GPO
Set-GPO	Modifies properties of a GPO
Remove-GPO	Deletes a GPO
Rename-GPO	Renames a GPO
Backup-GPO	Backs up one or more GPOs in a domain
Copy-GPO	Copies a GPO from one domain to another domain

Studijní materiály Okškolení

Cmdlet	Description
Restore-GPO	Restores a GPO from backup files
New-GPLink	Links a GPO to an AD DS container
Import-GPO	Imports GPO settings from a backed-up GPO
Set-GPRegistryValue	Configures one or more registry-based policy settings in a GPO

New-GPO requires only the *-Name* parameter, which must be unique in the domain in which you create the GPO. By default, the GPO is created in the domain of the user who is running the command. **New-GPO** also does not link the created GPO to an AD DS container.

The following command creates a new GPO from a starter GPO:

```
New-GPO -Name "IT Team GPO" -StarterGPOName "IT Starter GPO"
```

The following command links the new GPO to an AD DS container:

```
New-GPLink -Name "IT Team GPO" -Target "OU=IT,DC=adatum,DC=com"
```

Server Manager cmdlets

The **ServerManager** module for Windows

PowerShell contains cmdlets for managing server features, roles, and services. These cmdlets are the equivalent of the **Server Manager** user interface.

The **Server Manager** cmdlet names include the noun "WindowsFeature."

The following table lists the server management cmdlets.

Cmdlet	Description
Get-WindowsFeature	Gets information about Windows Server roles, services, and features on the local computer
Install-WindowsFeature	Installs roles, services, or features
Uninstall-WindowsFeature	Uninstalls roles, services, or features
Add-WindowsFeature	Windows Server 2008 R2 equivalent of Install-WindowsFeature
Remove-WindowsFeature	Windows Server 2008 R2 equivalent of Uninstall-WindowsFeature

```
Install-WindowsFeature "nla"
```

Cmdlet	Description
Get-WindowsFeature	Gets information about Windows Server roles, services, and features that are installed or are available for installation
Install-WindowsFeature	Installs one or more roles, services, or features
Uninstall-WindowsFeature	Uninstalls one or more roles, services, or features

The **Install-WindowsFeature** and **Uninstall-WindowsFeature** cmdlets require Windows Server 2012 R2 or newer, or Windows 8 or newer. In Windows Server 2008 R2 and Windows 7, the equivalent cmdlets are **Add-WindowsFeature** and **Remove-WindowsFeature**.



Note: On systems that support **Install-WindowsFeature** and **Uninstall-WindowsFeature**, the commands **Add-WindowsFeature** and **Remove-WindowsFeature** are still available as aliases that point to the newer commands.

The following command installs network load balancing on the local server:

```
Install-WindowsFeature "nlb"
```

Hyper-V cmdlets

Windows PowerShell offers more than 160 cmdlets for managing Hyper-V VMs, virtual hard disks, and other components of a Hyper-V environment. Hyper-V cmdlets are available in the Hyper-V module for Windows PowerShell.

You can install the Hyper-V module from within Windows PowerShell by installing the Windows feature. To do so, type the following command in the console, and then press Enter:

```
Enable-WindowsOptionalFeature -Feature Microsoft-Hyper-V-Management-PowerShell -Online
```

- More than 160 Hyper-V cmdlets
- Must install Hyper-V module by adding optional Windows features
- Cmdlets use one of three prefixes:
 - **VM**
 - **VHD**
 - **VFD**

Hyper-V cmdlets use one of three prefixes:

- “VM” for VM cmdlets
- “VHD” for Virtual hard disk cmdlets
- “VFD” for virtual hard disk cmdlets

The following table lists common cmdlets for managing Hyper-V VMs.

Cmdlet	Description
Get-VM	Gets properties of a VM
Set-VM	Sets properties of a VM
New-VM	Creates a new VM
Start-VM	Starts a VM
Stop-VM	Stops a VM
Restart-VM	Restarts a VM
Suspend-VM	Pauses a VM
Resume-VM	Resumes a paused VM
Import-VM	Imports a VM from a file

Cmdlet	Description
Export-VM	Exports a VM to a file
Checkpoint-VM	Creates a checkpoint of a VM

IIS management cmdlets

The Web server role is one of the most common server roles that administrators must manage. IIS cmdlets allow you to configure and manage application pools, websites, web applications, and virtual directories.

IIS management cmdlets are available in the **WebAdministration** module for Windows PowerShell and have the prefix "Web" in the noun part of their names. Cmdlets for managing application pools use the noun "WebAppPool," applications use the noun "WebApplication," and sites use the noun "WebSite."

The following table lists common IIS management cmdlets.

Cmdlet	Description
New-WebSite	Creates a new IIS website
Get-WebSite	Gets properties of an IIS website
Start-WebSite	Starts an IIS website
Stop-WebSite	Stops an IIS website
New-WebApplication	Creates a new web application
Remove-WebApplication	Deletes a web application
New-WebAppPool	Creates a new web application pool
Restart-WebAppPool	Restarts a web application pool

Cmdlet	Description
New-WebSite	Creates a new IIS website
Get-WebSite	Gets properties about an IIS website
Start-WebSite	Starts an IIS website
Stop-WebSite	Stops an IIS website
New-WebApplication	Creates a new web application
Remove-WebApplication	Deletes a web application
New-WebAppPool	Creates a new web application pool
Restart-WebAppPool	Restarts a web application pool

To create a new IIS website, type the following command in the console, and then press Enter:

```
New-WebSite "London" -PhysicalPath C:\inetpub\wwwroot\london -IPAddress 172.16.0.15  
-ApplicationPool LondonAppPool
```

 **Note:** The **WebAdministration** module represents IIS as a **PSDrive**, which you can navigate by using the **Set-Location IIS:** command. This allows you to navigate the IIS structure by using cmdlets such as **Get-ChildItem**. You will learn more about **PSDrives** in Module 5, "Using PSProviders and PSDrives."

Question: What Windows feature must you install before you can use Hyper-V cmdlets?

Lab: Windows administration

Scenario

You work for Adatum Corporation on the server support team. One of your first assignments is to configure the infrastructure service for a new branch office. Policy requires that you complete the tasks by using Windows PowerShell.

Objectives

After completing this lab, you will have:

- Created and managed Active Directory objects by using Windows PowerShell.
- Configured network settings on Windows Server by using Windows PowerShell.
- Created an IIS website by using Windows PowerShell.

Lab Setup

Estimated Time: **60 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you need to use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Exercise 1: Creating and managing Active Directory objects

Scenario

In this exercise, you will create and manage Active Directory objects to create an OU for a branch office, along with groups for OU administrators. You will create accounts for a user and computer in the branch office, in the default OU, and add the user to the administrators group. You will later move the user and computer to the OU that you created for the branch office. You will use individual Windows PowerShell commands to accomplish these tasks from a client computer.

The main tasks for this exercise are as follows:

1. Create a new organizational unit (OU) for a branch office.
 2. Create group for branch office administrators.
 3. Create a user and computer account for the branch office.
 4. Move the group, user, and computer accounts to the branch office OU.
- **Task 1: Create a new organizational unit (OU) for a branch office**
- From **LON-CL1**, use Windows PowerShell to create a new OU named **London**.
- **Task 2: Create group for branch office administrators**
- In the London OU, create the **London Admins** global security group.
- **Task 3: Create a user and computer account for the branch office**
1. In the **PowerShell** console, create a user account for the user **Ty Carlson**.
 2. Add the user to the **London Admins** group.
 3. Create a computer account for the **LON-CL2** computer.
- **Task 4: Move the group, user, and computer accounts to the branch office OU**
- Use Windows PowerShell to move the following group, user, and computer accounts to the London OU:
 - **London Admins**
 - **Ty Carlson**
 - **LON-CL2**

Results: After completing this exercise, you will have successfully identified and used commands for managing Active Directory objects in the Windows PowerShell command-line interface.

Exercise 2: Configuring network settings on Windows Server

Scenario

In this exercise, you will configure network settings on Windows Server. You will test network connectivity before and after, making changes to view the effect. You will use individual Windows PowerShell commands to accomplish these tasks on the server.

The main tasks for this exercise are as follows:

1. Test the network connection and view the configuration.
2. Change the server IP address.
3. Change the DNS settings and default gateway for the server.
4. Verify and test the changes.

► **Task 1: Test the network connection and view the configuration**

1. Switch to **LON-SVR1**.
2. Open **Windows PowerShell**.
3. Test the connection to **LON-DC1**, and then note the speed of the test.
4. View the network configuration for **LON-SVR1**.
5. Note the IP address, default gateway, and DNS server.

► **Task 2: Change the server IP address**

- Use Windows PowerShell to change the IP address for the **Ethernet** network interface to **172.16.0.15/16**.

► **Task 3: Change the DNS settings and default gateway for the server**

1. Change the DNS settings of the **Ethernet** network interface to point at **172.16.0.12**.
2. Change the default gateway for the **Ethernet** network interface to **172.16.0.2**.

► **Task 4: Verify and test the changes**

1. On **LON-SVR1**, verify the changes to the network configuration.
2. Test the connection to **LON-DC1**, and then note the difference in the test speed.

Results: After completing this exercise, you will have successfully identified and used Windows PowerShell commands for managing network configuration.

Exercise 3: Creating a website

Scenario

In this exercise, you will install the IIS server and create a new internal website for the London branch. You will use individual Windows PowerShell commands to accomplish these tasks on the server.

The main tasks for this exercise are as follows:

1. Install IIS on the server.
2. Create a folder on the server for the website files.
3. Create a new application pool for the website.
4. Create the IIS website.
5. Prepare for the next module.

► **Task 1: Install IIS on the server**

- Use Windows PowerShell to install IIS on **LON-SVR1**.

► **Task 2: Create a folder on the server for the website files**

- On **LON-SVR1**, use PowerShell to create a folder named **London** under **C:\inetpub\wwwroot** for the website files.

► **Task 3: Create a new application pool for the website**

- On **LON-SVR1**, use PowerShell to create an application pool for the site named **LondonAppPool**.

► **Task 4: Create the IIS website**

1. On **LON-SVR1**, use PowerShell to create the IIS website by using the following configuration:
 - o Name: **London**
 - o Physical path: The folder that you created earlier
 - o IP address: The current IP address of **LON-SVR1**
 - o Application pool: **LondonAppPool**
2. Open the website in Internet Explorer by using the IP address, and then verify that the site is using the provided settings.

 **Note:** Internet Explorer displays an error message. The error message details give the physical path of the site, which should be **C:\inetpub\wwwroot\london**.

Results: After completing this exercise, you will have successfully identified and used Windows PowerShell commands that would be used as part of a standardized Web server configuration.

► **Task 5: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Question: Are any other websites on the **LON-SVR1** server?

Module Review and Takeaways

Best Practice

Be sure to run the **Update-Help** command periodically so that you have the most up-to-date help for Windows PowerShell commands.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
When I run the Get-Help command for a cmdlet with the <i>-Example</i> parameter, I do not see any examples.	
I update the Windows PowerShell version on my system, but a new command does not appear to do anything.	

Review Questions

Question: What command in the Windows PowerShell command-line interface can you use instead of **ping.exe**?

Question: Name at least two ways in which you can create an Active Directory Domain Services (AD DS) user account by using Windows PowerShell.

Module 3

Working with the Windows PowerShell pipeline

Contents:

Module Overview	3-1
Lesson 1: Understanding the pipeline	3-2
Lesson 2: Selecting, sorting, and measuring objects	3-8
Lab A: Using the pipeline	3-16
Lesson 3: Filtering objects out of the pipeline	3-19
Lab B: Filtering objects	3-25
Lesson 4: Enumerating objects in the pipeline	3-28
Lab C: Enumerating objects	3-32
Lesson 5: Sending pipeline data as output	3-34
Lab D: Sending output to a file	3-39
Module Review and Takeaways	3-41

Module Overview

Windows PowerShell is not the first command-line shell to support the concept of a pipeline. For example, the command prompt in the Windows operating system supports a pipeline. However, the Windows PowerShell pipeline is more complex, more flexible, and more capable than that of older shells. The pipeline is a key concept and functional component of Windows PowerShell, and by mastering it, you can use Windows PowerShell more effectively and efficiently.



Additional Reading: For more information on the pipeline, refer to: "Understanding the Windows PowerShell Pipeline" at: <https://aka.ms/hj1a9f>

Objectives

After completing this module, you will be able to:

- Describe the purpose of the Windows PowerShell pipeline.
- Select, sort, and measure objects in the pipeline.
- Filter objects out of the pipeline.
- Enumerate objects in the pipeline.
- Send output consisting of pipeline data.

Lesson 1

Understanding the pipeline

In this lesson, you will learn about the Windows PowerShell pipeline and some basic techniques for running multiple commands in it. Running commands individually is both difficult and inefficient. Understanding how the pipeline works is fundamental to your success in using Windows PowerShell for administration.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the features and functionalities of the pipeline.
- Use the appropriate terminology to describe the pipeline output and pipeline objects.
- Explain how to discover and display object members.
- View object members.
- Describe the cmdlets used to format the pipeline output for display.
- Format pipeline output.

What is the pipeline?

Windows PowerShell runs commands in a pipeline. A *pipeline* is a chain of one or more commands in which the output from one command can pass as input to the next command. Even a single command runs in a pipeline. In Windows PowerShell, each command in the pipeline runs in sequence from left to right. For multiple commands, each command and its parameters are separated from the next command by a character known as a *pipe* (|). Specific rules dictate how output passes from one command to the next. You will learn about those rules in Module 4, "Understanding how the pipeline works."

- Windows PowerShell runs commands in a *pipeline*
- Each console command line is a pipeline
- Commands separated by a pipe character (|)
- Commands execute from left to right
- Output of each command is *piped* (passed) to the next
- The output of the last command in the pipeline is what appears on your screen
- Piped commands typically follow the pattern **Get |Set, Get | Where, or Select | Set**

As you interact with Windows PowerShell in the console host application, you should think of each command line as a single pipeline. You type a command or a series of commands and then press Enter to run the pipeline. The output of the last command in the pipeline appears on your screen. Another shell prompt follows that output, and you can enter commands into a new pipeline at that shell prompt.

 **Note:** You can type one logical command line over multiple physical lines in the console. For example, type **Get-Service**', and then press Enter. Windows PowerShell enters an extended prompt mode, which is indicated by the presence of two consecutive greater than signs (>>). This allows you to complete the command line. Press Ctrl+C to exit the command and return to the Windows PowerShell prompt.

In the previous modules, you learned about common actions, or *verbs*, associated with Windows PowerShell commands. When running multiple commands as part of a single pipeline, you most commonly see the verbs **Get** and **Set** used in combination. You use the output of a **Get-*** command as the input for a **Set-*** command. You often use these commands in combination with a filtering command, such as **Where** or **Select**. In that case, the output of **Get-*** is filtered by the **Where** or **Select** command before being piped to the **Set-*** command.

 **Note:** The **Where** command is an alias for **Where-Object**, and the **Select** command is an alias for **Select-Object**. Lesson 3, “Filtering objects out of the pipeline” discusses filtering in more detail.

Pipeline output

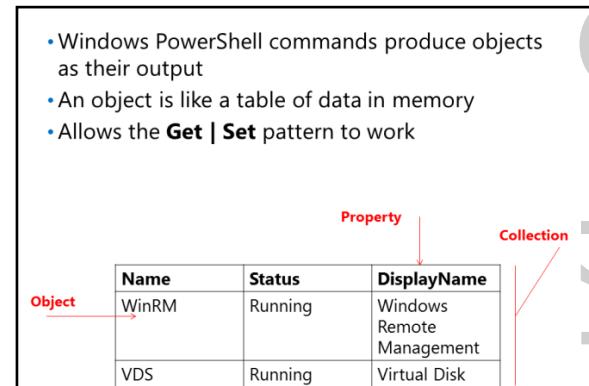
Windows PowerShell commands do not generate text as output. Instead, they generate objects. *Object* is a generic word that describes an in-memory data structure.

You can imagine command output as looking like a database table or a spreadsheet. In Windows PowerShell terminology, the table or spreadsheet consists of a collection of objects, or just a *collection* for short. Each row is a single object, and each column is a *property* of that object—that is, information about the object. For example, when you run the **Get-Service** command, it returns a collection of service objects. Each object has properties with names such as **Name**, **DisplayName**, and **Status**.

With its use of objects, Windows PowerShell differs greatly from other command-line shells in which the commands primarily generate text. In a text-based shell, suppose that you want to obtain a list of all the services that have been started. You might run a command to produce a text list of the services, with a different row for each service. Each row might contain the name of a service and some properties of the service, with each property separated by a comma or other character. To retrieve a particular property value, you would need to send that text output to another command that processes the text to pull out the particular value you need. That command would be written to understand the specific text format created by the first command. If the output of the first command ever changes, and the status information moves, you will have to rewrite the second command to get the new position information. Text-based shells require great skill with text parsing. This makes scripting languages such as Perl popular, because they offer strong text parsing and text manipulation features.

In Windows PowerShell, you tell Windows PowerShell to produce the collection of service objects and to then display only the **Name** property. The structure of the objects in memory enables Windows PowerShell to find the information for you, instead of you having to worry about the exact form of the command output.

This functionality makes it possible for the **Get | Set** pattern to work. Because the output of a **Get-*** command is an object, Windows PowerShell can find the properties needed by a **Set-*** command for it to work. You do not even need to tell Windows PowerShell which properties are needed.



Discovering object members

Members are the various components of an object and include:

- Properties. Describe attributes of the object. Examples of properties include a service name, a process ID number, and an event log message.
- Methods. Tell an object to perform an action. For example, a process object can quit itself, and an event log can clear itself.
- Events. Trigger when something happens to an object. A file might trigger an event when it opens, or a process might trigger an event when it has output to produce.

- Object members include:
 - Properties
 - Methods
 - Events
- Run a command that produces an object, and pipe that object to **Get-Member** to see a list of members
- Get-Member** is a discovery tool, similar to **Help**, that can help you learn to use the shell

Windows PowerShell primarily deals with properties and methods. For most commands that you run, the default on-screen output does not include all of an object's properties. Some objects have hundreds of properties, and the full list will not fit on the screen. Windows PowerShell includes several configuration files that list the object properties that should display by default. That is why you see three properties when you run **Get-Service**.

Use the **Get-Member** command to list all the members of an object. This command lists all the properties, even those that do not display on the screen by default. This command also lists methods and events and shows you the type name of the object. For example, the objects that **Get-Service** produces have the type name **System.ServiceProcess.ServiceController**. You can use the type name when you search the Internet to locate object documentation and examples. However, those examples are frequently in a programming language such as Microsoft Visual Basic or Microsoft Visual C#.

 **Note:** **Get-Member** has an alias: **gm**.

To use **Get-Member**, just pipe any command output to it. For example, type the following command in the console, and then press Enter:

```
Get-Service | Get-Member
```

 **Note:** The first command runs, produces its output, and then passes that output to **Get-Member**. Use caution when you run commands that might modify the system configuration, because those commands make changes to the system. You cannot use the **-WhatIf** parameter, which tells Windows PowerShell to only test and display the results of the command, when you want to pipe to **Get-Member**. The **-WhatIf** parameter prevents the command from producing any output. That means **Get-Member** receives no input and therefore does not display any output.

Demonstration: Viewing object members

In this demonstration, you will see how to run commands in the pipeline and how to use **Get-Member**.

Demonstration Steps

1. Sign in to **LON-CL1** as an administrator, and then start **Windows PowerShell**.
2. Display the members of the **Service** object.
3. Display the members of the **Process** object.
4. Display the list of members for the output of the **Get-ChildItem** command.

 **Note:** Note the default value for the **PSIsContainer** property and for the other returned members.

5. Display the list of members for the output of the **Get-ADUser** command.
 6. Display the list of members for the output of the **Get-ADUser** command, displaying all members.
-  **Note:** Note the number of returned properties and their names.

Formatting pipeline output

Windows PowerShell provides several ways to control the formatting of pipeline output. The default formatting of the output depends on the objects that exist in the output and the configuration files that define the output. After Windows PowerShell decides on the appropriate format, it passes the output to a set of formatting cmdlets without your input.

The formatting cmdlets are:

- **Format-List**
- **Format-Table**
- **Format-Wide**
- **Format-Custom**

• Use the following cmdlets to format pipeline output:

- **Format-List**
- **Format-Table**
- **Format-Wide**

• The **-Property** parameter:

- Is common to all formatting cmdlets
- Filters output to specified property names
- Can only specify properties that were passed to the formatting command

You can override the default output formatting by specifying any of the preceding cmdlets as part of the pipeline.

 **Note:** The **Format-Custom** cmdlet requires creating custom XML configuration files that define the format. It is used infrequently and is beyond the scope of this course.

Each formatting cmdlet accepts the *-Property* parameter. The *-Property* parameter accepts a comma-separated list of property names, and it then filters the list of properties that display and the order in which they appear. Keep in mind that when you specify property names for this parameter, those properties must have been returned by the original command—that is, the command that passed its output to the formatting cmdlet.

For example, the **Get-ADUser** cmdlet returns only a subset of properties, unless you specify its *-Properties* parameter. Therefore, if you specify the **City** property in the *-Property* parameter for a formatting cmdlet, it will appear as if the property is not set, unless you make sure that the **City** property is one of the properties returned for the users queried.

Some cmdlets default to passing a different set of properties for each formatting cmdlet. For example, the **Get-Service** cmdlet displays three properties (**Status**, **Name**, and **DisplayName**) in a table format by default. If you display the output of **Get-Service** as a list by using the **Get-Service | Format-List** command, six additional properties will display.

Format-List

The **Format-List** cmdlet, as the name suggests, formats the output of a command as a simple list of properties, where each property appears on a new line. If the command passing output to **Format-List** returns multiple objects, a separate list of properties for each object displays. List formatting is particularly useful when a command returns a large number of properties that would be hard to read in table format.

 **Note:** The alias for the **Format-List** cmdlet is **fl**.

To display a simple list in the console of the default properties for the processes running on the local computer, type the following command, and then press Enter:

```
Get-Process | Format-List
```

Format-Table

The **Format-Table** cmdlet formats output as a table, where each row represents an object, and each column represents a property. The table format is useful for displaying properties of many objects at the same time and comparing the properties of those objects.

By default, the table includes the property names as the column headers, which are separated from the data by a row of dashes. The formatting of the table depends on the returned objects. You can modify this formatting by using a variety of parameters, such as:

- *-AutoSize*. Adjusts the size and number of columns based on the width of the data. In Windows PowerShell 5.0 and newer, *-AutoSize* is set to **true** by default. In older versions of Windows PowerShell, the default values might truncate data in the table.
- *-HideTableHeaders*. Removes the table headers from the output.
- *-Wrap*. Causes text that is wider than the column width to wrap to the next line.

 **Note:** The alias for the **Format-Table** cmdlet is **ft**.

To display the **Name**, **ObjectClass**, and **Description** properties for all Windows Server Active Directory objects as a table, with the columns set to automatically size and wrap the text, type the following command in the console, and then press Enter:

```
Get-ADObject -filter * -Properties * | ft -Property Name, ObjectClass, Description -AutoSize -Wrap
```

Format-Wide

The output of the **Format-Wide** cmdlet is a single property in a single list displayed in multiple columns. This cmdlet functions like the */w* parameter of the **dir** command in **cmd.exe**. The wide format is useful for displaying large lists of simple data, such as the names of files or processes, in a compact format.

By default, **Format-Wide** displays its output in two columns. You can modify the number of columns by using the **-Column** parameter. The **-AutoSize** parameter, which works the same way it does for **Format-Table**, is also available. You cannot use **-AutoSize** and **-Column** together, however. The **-Property** parameter is also available, but in the case of **Format-Wide**, it can accept only one property name.

 **Note:** The alias for the **Format-Wide** cmdlet is **fw**.

To send the **DisplayName** property of all the Group Policy Objects (GPOs) in the current domain as output in three columns, type the following command in the console, and then press Enter:

```
Get-GPO -all | fw -Property DisplayName -Column 3.
```

Demonstration: Formatting pipeline output

In this demonstration, you will see how to format pipeline output.

Demonstration Steps

1. Display the services running on **LON-CL1** by using the default output format.
2. Display the names and statuses of the services running on **LON-CL1** in a simple list.
3. Display a list of the computers in the current domain, including the operating systems, by using the default output format.
4. Display a table that contains only the names and operating systems for all the computers in the current domain.
5. Display a list of all the Active Directory users in the current domain.
6. Display the user names of all the Active Directory users in the current domain. Display the list in a multicolumn format, and let Windows PowerShell decide the number of columns.

Question: Where can you find additional documentation about an object's members?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
The Format-Wide cmdlet accepts the -AutoSize and -Wrap parameters.	

Lesson 2

Selecting, sorting, and measuring objects

In this lesson, you will learn to manipulate objects in the pipeline by using commands that sort, select, and measure objects. Selecting, sorting, and measuring objects is essential to successfully creating automation in Windows PowerShell.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to sort objects by a specified property.
- Sort objects by using the **Sort-Object** command.
- Explain how to measure objects' numeric properties.
- Measure objects by using the **Measure-Object** command.
- Explain how to display a subset of objects in a collection.
- Explain how to display a customized list of objects' properties.
- Select objects by using the **Select-Object** command.
- Explain how to create calculated properties.
- Create custom calculated properties for display.

Sorting objects by a property

Some Windows PowerShell commands produce their output in a specific order. For example, the **Get-Process** and **Get-Service** commands produce output that is sorted alphabetically by name. **Get-EventLog** produces output that is sorted by time. In other cases, the output may not appear to be sorted at all. Sometimes, you may want command output sorted differently from the default. The **Sort-Object** command, which has the alias **sort**, can do that for you.

Sort-Object

The **Sort-Object** command accepts one or more property names to sort by. By default, the command sorts in ascending order. If you want to reverse the sort order, add the *-Descending* parameter. If you specify more than one property, the command first sorts by the first property, then by the second property, and so on. It is not possible in a single command to sort by one property in ascending order and another in descending order.

The following commands are all examples of sorting:

```
Get-Service | Sort-Object -Property Name -Descending
Get-Service | Sort Name -Desc
Get-Service | Sort Status,Name
```

- Each command determines its own default sort order
- **Sort-Object** can resort objects in the pipeline
- Here is an example:
 - **Get-Service | Sort-Object Name -Descending**
- Sorting enables grouping output by using:
 - The *-GroupBy* parameter
 - The **Group-Object** command

By default, string properties are sorted without regard to case. That is, lowercase and uppercase letters are treated the same. The parameters of **Sort-Object** allow you to specify a case-sensitive sort, a specific culture's sorting rules, and other options. As with other commands, you can view the help for **Sort-Object** for details and examples.

Grouping objects by property

Sorting objects also allows you to display objects in groups. The **Format-List**, **Format-Table**, and **Format-Wide** formatting cmdlets all have a *-GroupBy* parameter that accepts a property name. By using the *-GroupBy* parameter, you can group the output by the specified property. For example, the following command displays the names of services running on the local computer in two two-column lists that are grouped by the **Status** property:

```
Get-Service | Sort-Object Status,Name | fw -GroupBy Status
```

The *-GroupBy* parameter works similarly to the **Group-Object** command. The **Group-Object** command accepts piped input and gives you more control over the grouping of the objects. **Group-Object** has the alias **group**.

Demonstration: Sorting objects

In this demonstration, you will see how to sort objects by using the **Sort-Object** command.

Demonstration Steps

1. Display a list of processes.
2. Display a list of processes sorted by process ID.
3. Display a list of services sorted by status.
4. Display a list of services sorted in reverse order by status.
5. Display a list of the 10 most recent **Security** event log entries that is sorted with the newest entry first.
6. Display a list of the 10 most recent **Security** event log entries that is sorted with the oldest entry first, and then clear the event log.
7. Display the user names of all Active Directory users in wide format and sorted by surname.

Measuring objects

The **Measure-Object** command can accept any kind of object in a collection. By default, the command counts the number of objects in the collection and produces a measurement object that includes the count.



Note: The **Measure-Object** command has the alias **Measure**.

- **Measure-Object** accepts a collection of objects and counts them
- Add *-Property* to specify a single numeric property, and then add:
 - *-Average* to calculate an average
 - *-Minimum* to display the smallest value
 - *-Maximum* to display the largest value
 - *-Sum* to display the sum
- The output is a measurement object, and not whatever you piped in

```
Get-ChildItem -File | Measure -Property Length -Sum  
-Average -Minimum -Max
```

The **-Property** parameter of **Measure-Object** allows you to specify a single property, which must contain numeric values. You can then include the **-Sum**, **-Average**, **-Minimum**, and **-Maximum** parameters to calculate those aggregate values for the specified property.

 **Note:** Windows PowerShell allows you to *truncate* a parameter name, or use only a portion of the parameter name, if the truncated name clearly identifies the parameter. You will frequently see the **-Sum**, **-Minimum**, and **-Maximum** parameters written as **-Sum**, **-Min**, and **-Max**, corresponding to the common English abbreviations for those words. However, you cannot shorten **-Average** to **-Avg**, although beginners frequently try to. You can shorten the **-Average** parameter to **-Ave**, which is a valid truncation of the name.

The following command counts the number of files in a folder and displays the smallest, largest, and average file size:

```
Get-ChildItem -File | Measure -Property Length -Sum -Average -Minimum -Max
```

Demonstration: Measuring objects

In this demonstration, you will see how to measure objects by using the **Measure-Object** command.

Demonstration Steps

1. Display the number of services on your computer.
2. Display the number of Active Directory users.
3. Display the total amount and the average amount of virtual memory that the processes are using.

Selecting a subset of objects

Sometimes, you might not need to display all the objects that a command produces. For example, you have already seen that the **Get-EventLog** command has a parameter, **-Newest**, that produces a list of only the newest event log entries. Not all commands have the built-in ability to limit their output like that. However, the **Select-Object** command can do this for the output from any command.

 **Note:** The **Select-Object** command has the alias **Select**.

- One of two uses for **Select-Object**
- Use parameters to select the specified number of rows from the beginning or end of the piped-in collection:
 - **-First** for the beginning
 - **-Last** for the end
 - **-Skip** to skip a number of rows before selecting
 - **-Unique** to ignore duplicated rows
- You cannot specify any criteria for choosing specific rows

You can use **Select-Object** to select a subset of the objects that are passed to it in the pipeline. One of the ways you can do this is by position. If you think of a collection of objects as a table or spreadsheet, selecting a subset means selecting just certain rows. **Select-Object** can select a specific number of rows. You cannot tell **Select-Object** to choose those rows by column or property value but only by position. You can tell it to start from the beginning or the end of the collection. You can also tell it to skip a certain number of rows before the selection begins. You can use the **Sort-Object** command to control the row order before passing the objects to **Select-Object** in the pipeline.

For example, to select the first 10 processes based on the least amount virtual memory use, type the following command in the console, and then press Enter:

```
Get-Process | Sort-Object -Property VM | Select-Object -First 10
```

To select last 10 running services and sort them by name, type the following command in the console, and then press Enter:

```
Get-Service | Sort-Object -Property Name | Select-Object -Last 10
```

To select the five processes using the least amount of CPU, but skip the one process using the least CPU, type the following command in the console, and then press Enter:

```
Get-Process | Sort-Object -Property CPU -Descending | Select-Object -First 5 -Skip 1
```

Selecting unique objects

In one particular scenario, **Select-Object** looks at the properties and values when selecting the rows to return. If some members of the object collection passed to **Select-Object** have duplicate names and values, and you include the **-Unique** parameter, **Select-Object** returns only one of the duplicated objects.

For example, if you want to display user information for one user in each department, type the following command in the console, and then press Enter:

```
Get-ADUser -Filter * -Property Department | Sort-Object -Property Department | Select-Object -Unique
```

Selecting properties of objects

In addition to selecting the first or last number of rows from a collection, you can use **Select-Object** to specify the properties to display. You use the **-Property** parameter followed by a comma-separated list of the properties you want to display. If you think of a collection of objects as a table or spreadsheet, you are choosing the columns to display. After you choose the properties you want, **Select-Object** removes all the other properties (or columns). If you want to sort by a property but not display it, you first use **Sort-Object** and then use **Select-Object** to specify the properties you want to display.

- The second use of **Select-Object**
- Use the **-Property** parameter to specify a comma-separated list of properties (wildcards are accepted) to include
- You can combine the **-Property** parameter with **-First**, **-Last**, and **-Skip** to select a subset of rows

The following command displays a table containing the name, process ID, virtual memory size, paged memory size, and CPU usage for all the processes running on the local computer:

```
Get-Process | Select-Object -Property Name, ID, VM, PM, CPU | Format-Table
```

The **-Property** parameter works with the **-First** or **-Last** parameter. The following command returns the name and CPU usage for the 10 processes with the largest amount of CPU usage:

```
Get-Process | Sort-Object -Property CPU -Descending | Select-Object -Property Name, CPU -First 10
```

Use caution when specifying property names. Sometimes, the default screen display that Windows PowerShell creates does not use the real property names in the table column headers. For example, the output of **Get-Process** includes a column labeled **CPU(s)**. However, the **System.Diagnostics.Process** object type does not have a property that has that name. The actual property name is **CPU**. You can see this by piping the output of **Get-Process** to **Get-Member**.

 **Best Practice:** Always review the property names in the output of **Get-Member** before you use those property names in another command. By doing this, you can help to ensure that you use the actual property names and not ones created for display purposes.

Demonstration: Selecting objects

In this demonstration, you will see various ways to use the **Select-Object** command.

Demonstration Steps

1. Display 10 processes by largest amount of virtual memory use.
2. Display the current day of week—for example, Monday or Tuesday.
3. Display the 10 most recent **Security** event log entries. Include only the event ID, time written, and event message.
4. Display the names of the Active Directory computers grouped by operating system.

Creating calculated properties

Select-Object can create custom, or *calculated*, properties. Each of these properties has a label, or name, that Windows PowerShell displays in the same way it displays any built-in property name. Each calculated property also has an expression that defines the contents of the property. You create each calculated property by entering the values in a hash table.

What are hash tables?

A *hash table* is known in some other programming and scripting languages as an *associative array* or *dictionary*. One hash table can contain multiple items, and each item consists of a key and a value, or a *name-value pair*. Windows PowerShell uses hash tables many times and for many purposes. When you create your own hash table, you can specify your own keys.

- Calculated (custom) properties let you choose the output label and contents
- Each calculated property works like a single regular property in the property list accepted by **Select-Object**
- Create calculated properties by using a specific syntax:
 - **Label** defines the property name
 - **Expression** defines the property contents
 - Within the expression, **\$PSItem** (or **\$_**) refers to the piped-in object



 **Note:** Module 7, "Working with variables, arrays, and hash tables" covers hash tables in greater detail.

Select-Object hash tables

When you use a hash table to create calculated properties by using **Select-Object**, you must use the following keys that Windows PowerShell expects:

- **label, l, name, or n.** This specifies the label, or name, of the calculated property. Because the lowercase **l** resembles the number 1 in some fonts, try to use either **name, n, or label**.
- **expression or e.** This specifies the expression that sets the value of the calculated property.

 **Note:** This course will use **n** and **e** for most examples, because they use less space on the page than the other options.

This means that each hash table contains only one name-value pair. However, you can use more than one hash table to create multiple calculated properties.

For example, suppose that you want to display a list of processes that includes the name, ID, virtual memory use, and paged memory use of each process. You want to use the column labels **VirtualMemory** and **PagedMemory** for the last two properties, and you want those values displayed in bytes. To do so, type the following command in the console, and then press Enter:

```
Get-Process |  
Select-Object  
Name, ID, @{n='VirtualMemory'; e={$PSItem.VM}}, @{n='PagedMemory'; e={$PSItem.PM}}
```

 **Note:** You can type the command exactly as shown and press Enter after the vertical pipe character. When you do so, you enter the extended prompt mode. After you type the rest of the command, press Enter on a blank line to run the command.

The previous command includes two hash tables, and each one creates a calculated property. The hash table might be easier to interpret if you write it a bit differently:

```
@{  
    n='VirtualMemory';  
    e={ $PSItem.VM }  
}
```

The previous example uses a semicolon to separate the two key-value pairs, and it uses the keys **n** and **e**. Windows PowerShell expects these keys. The label (or name) is just a string and is therefore enclosed in single quotation marks. Windows PowerShell accepts either quotation marks (" ") or single quotation marks (' ') for this purpose. The expression is a small piece of executable code called a *script block* and is contained within braces ({}).

 **Note:** **\$PSItem** is a special variable created by Windows PowerShell. It represents whatever object was piped into the **Select-Object** command. In the previous example, that is a **Process** object. The period after **\$PSItem** lets you access a single member of the object. In this example, one calculated property uses the **VM** property, and the other uses the **PM** property.

 **Note:** Windows PowerShell 1.0 and Windows PowerShell 2.0 used **\$_** instead of **\$PSItem**. That older syntax is compatible in Windows PowerShell 3.0 and newer, and many experienced users continue to use it out of habit.

A comma separates each property—that is, **Name**, **ID**, and each calculated property—from the others in the list.

Formatting calculated properties

You might want to modify the previous command to display the memory values in megabytes (MB). Windows PowerShell understands the abbreviations **KB**, **MB**, **GB**, **TB**, and **PB** as representing kilobytes, megabytes, gigabytes, terabytes, and petabytes, respectively. Therefore, you might modify your command as follows:

```
Get-Process |
Select-Object Name,
ID,
@{n='VirtualMemory(MB)';e={$PSItem.VM / 1MB}},
@{n='PagedMemory(MB)';e={$PSItem.PM / 1MB}}
```

In addition to the revised formatting that makes the command easier to read, this example changes the column labels to include the **MB** designation. It also changes the expressions to include a division operation, dividing each memory value by 1 MB. However, the resulting values have several decimal places, which is unattractive.

To improve the output, make the following change:

```
Get-Process |
Select-Object Name,
ID,
@{n='VirtualMemory(MB)';e='{0:N2}' -f ($PSItem.VM / 1MB)},
@{n='PagedMemory(MB)';e='{0:N2}' -f ($PSItem.PM / 1MB)}
```

This example uses the Windows PowerShell **-f** formatting operator. When used with a string, the **-f** formatting operator tells Windows PowerShell to replace one or more placeholders in the string with the specified values that follow the operator.

In this example, the string that precedes the **-f** operator tells Windows PowerShell what data to display. The string '**{0:N2}**' signifies displaying the first data item as a number with two decimal places. The original mathematical expression comes after the operator. It is in parentheses to make sure that it runs as a single unit. You can type this command exactly as shown, press Enter on a blank line, and then view the results.

The syntax in the previous example might seem confusing, because it contains a lot of punctuation symbols. Start with the basic expression:

```
'{0:N2}' -f ($PSItem.VM / 1MB)
```

The previous expression divides the **VM** property by 1 MB and then formats the result as a number having up to two decimal places. That expression is then placed into the hash table:

```
@{n='VirtualMemory(MB)';e='{0:N2}' -f ($PSItem.VM / 1MB) }}
```

That hash table creates the custom property named **VirtualMemory(MB)**.



Note: You can read more about the **-f** operator by running **Help About_Operators** in Windows PowerShell.

Demonstration: Creating calculated properties

In this demonstration, you will see how to use **Select-Object** to create calculated properties. You will then see how those calculated properties behave like regular properties.

Demonstration Steps

1. On **LON-CL1**, display a list of Active Directory user accounts and their creation dates.
2. View the same list sorted by creation date, from newest to oldest.
3. View a list of the same users in the same order but displaying the user name and the age of the account, in days.

 **Note:** To complete the last step, you will use the **New-TimeSpan** cmdlet to calculate the account age.

4. View the same list, but sorting the results at the end, instead of the middle of the command.

Question: Why might you use the *-First* parameter of **Select-Object**?

Lab A: Using the pipeline

Scenario

You must produce several basic management reports that include specified information about the computers in your environment. You will create the reports by using the Windows PowerShell commands you have learned in this lesson to organize and format the information displayed.

Objectives

After completing this lab, you will be able to modify and sort pipeline objects.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Selecting, sorting, and displaying data

Scenario

In this exercise, you will produce lists of management information from the computers in your environment. For each task, you will discover the necessary commands and use **Select-Object**, **Sort-Object**, and the formatting cmdlets to customize the final output of each command.

The main tasks for this exercise are as follows:

1. Display the current day of the year.
2. Display information about installed hotfixes.
3. Display a list of available scopes from the DHCP server.
4. Display a sorted list of enabled Windows Firewall rules.
5. Display a sorted list of network neighbors.
6. Display information from the DNS name resolution cache.
7. Prepare for the next lab.

► **Task 1: Display the current day of the year**

1. On **LON-CL1**, start Windows PowerShell with administrative credentials.
2. Using a keyword such as **date**, find a command that can display the current date.
3. Display the members of the object produced by the command that you found in the previous step.
4. Display only the day of the year.
5. Display the results of the previous command on a single line.

► **Task 2: Display information about installed hotfixes**

1. Using a keyword such as **hotfix**, find a command that can display a list of the installed hotfixes.
2. Display the members of the object produced by the command that you found in the previous step.
3. Display a list of the installed hotfixes. Display only the installation date, hotfix ID number, and name of the user who installed the hotfix.
4. Display a list of the installed hotfixes. Display only the hotfix ID, the number of days since the hotfix was installed, and the name of the user who installed the hotfix.

► **Task 3: Display a list of available scopes from the DHCP server**

1. Using a keyword such as **DHCP** or **scope**, find a command that can display a list of Internet Protocol version 4 (IPv4) Dynamic Host Configuration Protocol (DHCP) scopes.
2. View the help for the command.
3. Display a list of the available IPv4 DHCP scopes on **LON-DC1**.
4. Display a list of the available IPv4 DHCP scopes on **LON-DC1**. This time, include only the scope ID, subnet mask, and scope name, and display the data in a single column.

► **Task 4: Display a sorted list of enabled Windows Firewall rules**

1. Using a keyword such as **rule**, find a command that can display the firewall rules.
2. Display a list of the firewall rules.
3. View the help for the command that displays the firewall rules.
4. Display a list of the firewall rules that are enabled.
5. Display the same data in a table, making sure no information is truncated.
6. Display a list of the enabled firewall rules. Display only the rules' display names, the profiles they belong to, their directions, and whether they allow or deny access.
7. Sort the list in alphabetical order first by profile and then by display name, with the results appearing in a separate table for each profile.

► **Task 5: Display a sorted list of network neighbors**

1. Using a keyword such as **neighbor**, find a command that can display the network neighbors.
2. View the help for the command.
3. Display a list of the network neighbors.
4. Display a list of the network neighbors that is sorted by state.
5. Display a list of the network neighbors that is grouped by state, displaying only the IP address in as compact a format as possible and letting Windows PowerShell decide how to optimize the layout.

► **Task 6: Display information from the DNS name resolution cache**

1. Test your network connection to both **LON-DC1** and **LON-CL1** so that you know the Domain Name System (DNS) client cache is populated with data.
2. Using a keyword such as **cache**, find a command that can display items from the DNS client cache.
3. Display the DNS client cache.
4. Display the DNS client cache. Sort the list by record name, and display only the record name, record type, and Time to Live. Use only one column to display all the data.

Results: After completing this exercise, you should have produced several custom reports that contain management information from your environment.

► **Task 7: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Question: Suppose that you want to produce output that includes all of an object's properties except one. What is the most efficient way to do that?

Question: List the basic formatting commands, and explain why you might use each one.

Lesson 3

Filtering objects out of the pipeline

In this lesson, you will learn how to filter objects out of the pipeline by using the **Where-Object** cmdlet to specify various criteria. This differs from the ability of **Select-Object** to select several objects from the beginning or end of a collection, and it is more flexible. With this new technique, you will be able to keep or remove objects based on criteria of almost any complexity.

Lesson Objectives

After completing this lesson, you will be able to:

- List the major Windows PowerShell comparison operators.
- Explain how to filter objects by using basic syntax.
- Explain how to filter objects by using advanced syntax.
- Filter objects.
- Explain how to optimize filtering performance in the pipeline.

Comparison operators

To start filtering, you need a way to tell Windows PowerShell which objects you want to keep and which ones you want to remove from the pipeline. You can do this by specifying criteria for the objects that you want to keep. You do so by using one of the Windows PowerShell comparison operators to compare a particular property of an object to a value that you specify. Windows PowerShell keeps the object if the comparison is true, and removes it if the comparison is false.

The following table lists the basic comparison operators and what they mean.

Comparison type	Case-insensitive operator	Case-sensitive operator
Equality	-eq	-ceq
Inequality	-ne	-cne
Greater than	-gt	-cgt
Less than	-lt	-clt
Greater than or equal to	-ge	-cge
Less than or equal to	-le	-cle
Wildcard equality	-like	-clike

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-le	Less than or equal to
-ge	Greater than or equal to

These operators are all case insensitive when used with strings. This means that the results will be the same whether letters are capitalized or not. A case-sensitive version of each operator is available and begins with the letter **c**, such as **-ceq** and **-cne**.

Studijní materiály Okškolení

Windows PowerShell also contains the **-like** operator and its case-sensitive companion, **-clike**. The **-like** operator resembles **-eq** but supports the use of the question mark (?) and asterisk (*) wildcard characters in string comparisons.

Additional, more-advanced operators exist that are beyond the scope of this course. These operators include the following:

- The **-in** and **-contains** operators, which test whether an object exists in a collection.
- The **-as** operator, which tests whether an object is of a specified type.
- The **-match** and **-cmatch** operators, which compare a string to a regular expression.

Windows PowerShell also contains many operators that reverse the logic of the comparison, such as **-notlike** and **-notin**.

You can make comparisons directly at the command prompt, which returns either True or False. Here is an example:

```
PS C:\> 100 -gt 10
True
PS C:\> 'hello' -eq 'HELLO'
True
PS C:\> 'hello' -ceq 'HELLO'
False
```

This technique makes it easy to test comparisons before you use them in a command.

Basic filtering syntax

The **Where-Object** command (and its alias, **Where**) has two syntax forms: basic and advanced. These two forms are defined by an extensive list of parameter sets—one for each comparison operator. This allows for an easy-to-read syntax, especially in the basic form. For example, to display a list of only the running services, type the following command in the console, and then press Enter:

```
Get-Service | Where Status -eq Running
```

- The **Where-Object** command provides filtering
- Here are examples of basic syntax:

```
Get-Service |
Where Status -eq Running

Get-Process |
Where CPU -gt 20
```

You start this basic syntax with the **Where** alias (you can also specify the full command name, **Where-Object**), followed by the property that you want Windows PowerShell to compare. Then you specify a comparison operator, followed by the value that you want Windows PowerShell to compare. Every object that has the specified value in the specified property will be kept.

If you misspell the property name, or if you provide the name of a nonexistent property, Windows PowerShell will not generate an error. Instead, your command will not generate any output. For example, consider the following command:

```
Get-Service | Where Stat -eq Running
```

This command produces no output, because no service object has a **Stat** property that contains the value **Running**. In fact, none of the service objects has a **Stat** property. The comparison returns **False** for every object, which filters out all objects from the results.



Note: Because of the large number of parameter sets needed to make the basic syntax functional, the help file for **Where-Object** is very long and might be difficult to read. Consider skipping the initial syntax section and going directly to the description or examples if you need help with this command.

Limitations of the basic syntax

You can use the basic syntax for only a single comparison. You cannot, for example, display a list of the services that are stopped and have a start mode of **Automatic**, because that requires two comparisons.

You cannot use the basic syntax with complex expressions. For example, the **Name** property of a service object consists of a string of characters. Windows PowerShell uses a **System.String** object to contain that string of characters, and a **System.String** object has a **Length** property. The following command will not work with the basic filtering syntax:

```
Get-Service | Where Name.Length -gt 5
```

The intent is to display all the services that have a name longer than five characters. However, this command will never produce output. As soon as you exceed the capabilities of the basic syntax, you must move to the advanced filtering syntax.

Advanced filtering syntax

The advanced syntax of **Where-Object** uses a filter script. A *filter script* is a script block that contains the comparison and that you pass by using the **-FilterScript** parameter. Within that script block, you can use the built-in **\$PSItem** variable (or **\$_**, which is also valid in versions of Windows PowerShell before 3.0) to reference whatever object was piped into the command. Your filter script runs one time for each object that is piped into the command. When the filter script returns True, that object is passed down the pipeline as output. When the filter script returns False, that object is removed from the pipeline.

- Supports multiple conditions and has no restrictions on what kinds of expressions you can use
- Requires a filter script that contains your filtering criteria and that evaluates to either True or False
- Inside the filter script, use **\$PSItem** or **\$_** to refer to the object that was piped into the command



The following two commands are functionally identical. The first uses the basic syntax, and the second uses the advanced syntax to do the same thing:

```
Get-Service | Where Status -eq Running
Get-Service | Where-Object -FilterScript { $PSItem.Status -eq 'Running' }
```

The **-FilterScript** parameter is positional, and most users omit it. Most users also use the **Where** alias or the **? alias**, which is even shorter. Experienced Windows PowerShell users also use the **\$_** variable instead of **\$PSItem**, because only **\$_** is allowed in Windows PowerShell 1.0 and Windows PowerShell 2.0. The following commands perform the same task as the previous two commands:

```
Get-Service | Where {$PSItem.Status -eq 'Running'}
Get-Service | ? {$_.Status -eq 'Running'}
```

The single quotation marks (' ') around **Running** in these examples are required to make it clear that it is a string. Otherwise, Windows PowerShell will try to run a command called **Running**, which will fail because no such command exists.

Combining multiple criteria

The advanced syntax allows you to combine multiple criteria by using the **-and** and **-or** Boolean, or logical, operators. Here is an example:

```
Get-EventLog -LogName Security -Newest 100 |  
Where { $PSItem.EventID -eq 4672 -and $PSItem.EntryType -eq 'SuccessAudit' }
```

The logical operator must have a complete comparison on either side of it. In the preceding example, the first comparison checks the **EventID** property, and the second comparison checks the **EntryType** property. The following example is one that many beginning users try. It is incorrect, because the second comparison is incomplete:

```
Get-Process | Where { $PSItem.CPU -gt 30 -and VM -lt 10000 }
```

The problem is that **VM** has no meaning, but **\$PSItem.VM** would be correct. Here is another common mistake:

```
Get-Service | Where { $PSItem.Status -eq 'Running' -or 'Starting' }
```

The problem is that '**Starting**' is not a complete comparison. It is just a string value, so **\$PSItem.Status -eq 'Starting'** would be the correct syntax for the intended result.

Accessing properties that contain True or False

Remember that the only purpose of the filter script is to produce a True or False value. Usually, you produce those values by using a comparison operator, like in the examples that you have seen up to this point. However, when a property already contains either True or False, you do not have to explicitly make a comparison. For example, the objects produced by **Get-Process** have a property named **Responding**. This property contains either True or False. This indicates whether the process represented by the object is currently responding to the operating system. To obtain a list of the processes that are responding, you can use either of the following commands:

```
Get-Process | Where { $PSItem.Responding -eq $True }  
Get-Process | Where { $PSItem.Responding }
```

In the first command, the special shell variable **\$True** is used to represent the Boolean value True. The second command has no comparison at all, but it works because the **Responding** property already contains True or False.

It looks similar to reverse the logic to list only the processes that are not responding:

```
Get-Process | Where { -not $PSItem.Responding }
```

In the preceding example, the **-not** logical operator changes True to False, and it changes False to True. Therefore, if a process is not responding, its **Responding** property is False. The **-not** operator changes the result to True, which causes the process to be passed into the pipeline and included in the final output of the command.

Accessing properties without limitations

Although the basic filtering syntax can access only the direct properties of the object being evaluated, the advanced syntax does not have that limitation. For example, to display a list of all the services that have names longer than eight characters, use this:

```
Get-Service | Where {$PSItem.Name.Length -gt 8}
```

Demonstration: Filtering

In this demonstration, you will see various ways to filter objects out of the pipeline.

Demonstration Steps

1. On **LON-CL1**, use basic filtering syntax to display a list of the Server Message Block (SMB) shares that include a dollar sign (\$) in the share name.
2. Use advanced filtering syntax to display a list of the physical disks that are in healthy condition, displaying only their names and statuses.
3. Display a list of the disk volumes that are fixed disks and that use the NTFS file system. Display only the drive letter, drive label, drive type, and file system type. Display the data in a single column.
4. Using advanced filtering syntax but without using the **\$PSItem** variable, display a list of the Windows PowerShell command verbs that begin with the letter C. Display only the verb names in as compact a format as possible.

Optimizing filtering performance

The way you write your commands can have a significant effect on performance. Imagine that you have a container of plastic blocks. Each block is red, green, or blue. Each block has a letter of the alphabet printed on it. You have to put all the red blocks in alphabetical order. What would you do first?

Consider writing this task as a Windows PowerShell command by using the fictional **Get-Block** command. Which of the following two examples do you think will be faster?

```
Get-Block | Sort-Object -Property Letter |
Where-Object -FilterScript { $PSItem.Color -eq 'Red' }
Get-Block | Where-Object -FilterScript { $PSItem.Color -eq 'Red' } | Sort-Object -Property Letter
```

The second command will be faster, because it removes unwanted blocks from the pipeline so that only the remaining blocks are sorted. The first command sorts all the blocks and then removes many of them. This means that much of the sorting effort was wasted.

Many Windows PowerShell scripters use a *mnemonic*, which is a phrase that serves as a simple reminder, to help them remember to do the correct thing when they are optimizing performance. The phrase is *filter left*, and it means that any filtering should occur as far to the left, or as close to the beginning of the command line, as possible.

Sometimes, moving filtering as far to the left as possible means that you will not use **Where-Object**. For example, the **Get-ChildItem** command can produce a list that includes files and folders. Each object produced by the command has a property named **PSIsContainer**. It contains True if the object represents a folder and False if the object represents a file. The following command will produce a list that includes only files:

```
Get-ChildItem | Where { -not $PSItem.PSIsContainer }
```

- To improve performance, move filtering as close to the beginning of the command line as possible
- Some commands have parameters that can do some filtering for you, so whenever possible, use those parameters instead of **Where-Object**

However, this is not the most efficient way to produce the result. The **Get-ChildItem** command has a parameter that limits the command's output:

```
Get-ChildItem -File
```

When it is possible, check the help files for the commands that you use to see whether they contain a parameter that can do the filtering you want.

Question: Do you find `$_` or `$PSItem` easier to remember and use?

Question: Is the following command the most efficient way to produce a list of services that have names beginning with `svc`?

```
Get-Service | Where Name -like svc*
```

Lab B: Filtering objects

Scenario

You are continuing your project to create management reports by using Windows PowerShell. You have to retrieve additional management information about the computers in your environment. You need the output of your commands to include only specified information and objects.

Objectives

After completing this lab, you will be able to filter objects out of the pipeline by using basic and advanced syntax forms.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - o User name: **Administrator**
 - o Password: **Pa55w.rd**
 - o Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Filtering objects

Scenario

In this exercise, you will use filtering to produce lists of management information that include only specified data and elements for the reports you must produce.

The main tasks for this exercise are as follows:

1. Display a list of all the users in the Users container of Active Directory.
2. Create a report showing the Security event log entries that have the event ID 4624.
3. Display a list of the encryption certificates installed on the computer.
4. Create a report that shows the disk volumes that are running low on space.
5. Create a report that displays specified Control Panel items.
6. Prepare for the next lab.

► **Task 1: Display a list of all the users in the Users container of Active Directory**

1. On **LON-CL1**, open **Windows PowerShell** as an administrator.
2. Using a keyword such as **user**, find a command that can list Active Directory users.
3. View the help for the command, and identify any mandatory parameters.
4. Display a list of all the users in Active Directory in a format that lets you easily compare properties.
5. Display the same list of all the users in the same format. This time, however, display only those users in the Users container of Active Directory. Use a search base of "**cn=Users,dc=adatum,dc=com**" for this task.

► **Task 2: Create a report showing the Security event log entries that have the event ID 4624**

1. Display only the total number of **Security** event log entries that have the event ID 4624.
2. Display the full list of the **Security** event log entries that have the event ID 4624, and show only the time written, event ID, and message.
3. Display only the 10 oldest entries in a format that lets you view the message details.

► **Task 3: Display a list of the encryption certificates installed on the computer**

1. Display a directory listing of all the items on the **CERT** drive. Include subfolders in the list.
2. Display the list, again and show the name and issuer for only the certificates that do not have a private key. Show the results in one column.
3. Display the list, again and show only the current certificates. Those certificates have a **NotBefore** date that is before today and a **NotAfter** date that is after today. Include the **NotBefore** and **NotAfter** properties in the results, and display the results in a format that allows you to easily compare dates. Also, make sure that no data is truncated.

► **Task 4: Create a report that shows the disk volumes that are running low on space**

1. Display a list of the disk volumes.
2. Display a list in one column of the volumes that have more than zero bytes of free space.
3. Display a list of the volumes that have less than 99 percent free space and more than zero bytes of free space. Show only the drive letter and disk size, in MB.
4. Display a list of the volumes that have less than 10 percent free space and more than zero bytes of free space. This command might produce no results if no volumes on your computer meet the criteria.



Note: This command might not produce any output on your lab computer if the computer has more than 10 percent free space on each of its volumes.

► **Task 5: Create a report that displays specified Control Panel items**

1. Display a list of all the Control Panel items.
2. Display the names and descriptions, sorted by name, of the Control Panel items in the **System and Security** category.
3. Display the same list, excluding any Control Panel items that exist in more than one category. Make sure the command performance is optimized.

Results: After completing this exercise, you should have used filtering to produce lists of management information that include only specified data and elements.

► **Task 6: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Question: Do you prefer the basic or advanced syntax of **Where-Object**?

Question: What is the difference between **Select-Object** and **Where-Object**?

Question: In the first task of this lab, were you able to achieve the goal without using the **Where-Object** command?

Lesson 4

Enumerating objects in the pipeline

In this lesson, you will learn how to enumerate objects in the pipeline so that you can work with one object at a time during automation. Enumerating objects builds on the skills you already learned, and it is a building block for creating automation scripts.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of enumeration.
- Explain how to enumerate objects by using basic syntax.
- Perform basic enumerations.
- Explain how to enumerate objects by using advanced syntax.
- Perform advanced enumeration.

Purpose of enumeration

Enumeration is the process of performing a task on each object, one at a time, in a collection.

Frequently, Windows PowerShell does not require you to explicitly enumerate objects. For example, if you need to stop every running Notepad process on your computer, you can run either of these two commands:

```
Get-Process -Name Notepad | Stop-Process  
Stop-Process -Name Notepad
```

One common scenario that requires enumeration is when you run a method of an object, and no command provides the same functionality as that method. For example, the objects produced by **Get-Process** have a **Start** method that starts the process. However, you might never use that method, because the **Start-Process** command does the same thing.

Consider the kind of object produced when you run **Get-ChildItem -File** on a disk drive. This object type, **System.IO.FileInfo**, has a method named **Encrypt** that can encrypt a file by using the current user account's encryption certificate. No equivalent command is built in to Windows PowerShell, so you might have to run that method on many file objects that you wanted to encrypt. Enumeration allows you to do this with a single command.

- To take a collection of objects and:
 - Run an action on each item
 - Process them one at a time
- Not necessary when Windows PowerShell has a command that can perform the action you need
- Useful when an object has a method that does what you want, but Windows PowerShell does not offer an equivalent command

Basic enumeration syntax

The **ForEach-Object** command performs enumeration. It has two common aliases: **ForEach** and **%**. Like **Where-Object**, **ForEach-Object** has a basic syntax and an advanced syntax.

In the basic syntax, you can run a single method or access a single property of the objects that were piped into the command. Here is an example:

```
Get-ChildItem -Path C:\Encrypted\ -File |  
ForEach-Object -MemberName Encrypt
```

```
Get-ChildItem -Path C:\Example -File |  
ForEach-Object -MemberType Encrypt  
  
Get-ChildItem -Path C:\Example -File |  
ForEach Encrypt  
  
Get-ChildItem -Path C:\Example -File |  
% -MemberType Encrypt
```



With this syntax, you do not include the parentheses after the member name if the member is a method. Because this basic syntax is meant to be short, you will frequently see it written without the **-MemberName** parameter name, and you might see it written with an alias instead of the full command name. For example, both of the following commands perform the same action:

```
Get-ChildItem -Path C:\Encrypted\ -File | ForEach Encrypt  
Get-ChildItem -Path C:\Encrypted\ -File | % Encrypt
```



Note: You might not discover many scenarios where you must use enumeration. Each new operating system introduces new Windows PowerShell commands. Windows 8, Windows Server 2012, and newer operating systems introduced thousands of commands. Many of those new commands perform actions that previously required enumeration.

Limitations of the basic syntax

The basic syntax can access only a single property or method. It cannot perform logical comparisons that use **-and** or **-or**; it cannot make decisions; and it cannot run any other commands or code. For example, the following command does not run, and it produces an error:

```
Get-Service | ForEach -MemberName Stop -and -MemberName Close
```

Demonstration: Basic enumeration

In this demonstration, you will see how to use the basic enumeration syntax to enumerate several objects in a collection.

Demonstration Steps

1. Display only the name of every service installed on the computer.
2. Use enumeration to clear the **System** event log.

Advanced enumeration syntax

The advanced syntax for enumeration provides more flexibility and functionality than the basic syntax. Instead of letting you access a single object member, you can run a whole script. That script can include just one command, or it can include many commands in sequence.

For example, to encrypt a set of files by using the advanced syntax, type the following command in the console, and then press Enter:

- Allows you to perform any task by writing commands in a script block
- Uses **\$PSItem** or **\$_** to reference the objects that were piped into the command:
 - `Get-ChildItem C:\Test -File | ForEach-Object { $PSItem.Encrypt() }`
- Has additional parameters that allow you to specify actions to take before and after the collection of objects is processed

```
Get-ChildItem -Path C:\ToEncrypt\ -File | ForEach-Object -Process { $PSItem.Encrypt() }
```

The **ForEach-Object** command can accept any number of objects from the pipeline. It has the **-Process** parameter that accepts a script block. This script block runs one time for each object that was piped in. Every time that the script block runs, the built-in variable **\$PSItem** (or **\$_**) can be used to refer to the current object. In the preceding example command, the **Encrypt()** method of each file object runs.



Note: When used with the advanced syntax, method names are always followed by opening and closing parentheses, even when the method does not have any input arguments. For methods that do need input arguments, provide them as a comma-separated list inside the parentheses. Do not include a space or other characters between the method name and the opening parenthesis.

Advanced techniques

In some situations, you might need to repeat a particular task a specified number of times. You can use **ForEach-Object** for that purpose when you pass it an input that uses the range operator. The *range operator* is two periods (..) with no space between them. For example, run the following command:

```
1..100 | ForEach-Object { Get-Random }
```

In the preceding command, the range operator produces integer objects from 1 through 100. Those 100 objects are piped to **ForEach-Object**, forcing the script block to run 100 times. However, because neither **\$_** nor **\$PSItem** appear in the script block, the actual integers are not used. Instead, the **Get-Random** command runs 100 times. The integer objects are used only to set the number of times the script block runs.

Demonstration: Advanced enumeration

In this demonstration, you will see two ways to use the advanced enumeration syntax to perform tasks on several objects.

Demonstration Steps

1. Modify all the items in the **HKEY_CURRENT_USER\Network** subkey so that all the names are uppercase.
2. Create a directory named **Test** in all the **Democode** folders on the **Allfiles** drive, and display the path for each directory.

Question: If you have programming or scripting experience, does **ForEach-Object** look familiar to you?

Lab C: Enumerating objects

Scenario

You are a network administrator for Adatum Corporation. You were recently made responsible for managing the infrastructure for the London branch office. You have to complete several management tasks by using Windows PowerShell. These tasks require you to perform actions on multiple objects.

Objectives

After completing this lab, you will be able to enumerate pipeline objects by using both basic and advanced syntax.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Enumerating objects

Scenario

In this exercise, you will write commands that manipulate multiple objects in the pipeline. In some tasks, you need to use enumeration. In other tasks, you will not need to use enumeration. Decide the best approach for each task.

The main tasks for this exercise are as follows:

1. Display a list of files on the E: drive of your computer.
2. Use enumeration to produce 100 random numbers.
3. Run a method of a Windows Management Instrumentation (WMI) object.
4. Prepare for the next lab.

► **Task 1: Display a list of files on the E: drive of your computer**

1. On **LON-CL1**, open **Windows PowerShell** as an administrator.
2. Display a directory listing of all the items on the **E** drive. Include subfolders in the list.
3. Display a list of all the files on the E: drive, without displaying directory names.

► **Task 2: Use enumeration to produce 100 random numbers**

1. Using a keyword such as **random**, find a command that produces random numbers.
2. View the help for the command.
3. Run **1..100** to put 100 numeric objects into the pipeline.
4. Run the command again. For each numeric object, produce a random number that uses the numeric object as the seed.

► **Task 3: Run a method of a Windows Management Instrumentation (WMI) object**

1. Close all applications other than the **Windows PowerShell** console.
2. Run the command **Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges**.
3. Display the members of the object produced by the previous command.
4. In the member list, find a method that restarts the computer.
5. Run the command again, and use enumeration to run the method that restarts the computer.



Note: The command will restart the machine you run it on.

Results: After completing this exercise, you should have written commands that manipulate multiple objects in the pipeline.

► **Task 4: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Question: Do you prefer the basic or advanced syntax of **ForEach-Object**?

Lesson 5

Sending pipeline data as output

When you provide information about your network infrastructure, it is often a requirement that you provide the information in specific formats. This might mean using a format for displaying on the screen, for printing a hard copy, or for storing in a file for later use. In this lesson, you will learn how to send pipeline data to files and in various output formats.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to write pipeline data to a file.
- Explain how to convert pipeline data to the comma-separated values (CSV) format.
- Explain how to convert pipeline data to the XML format.
- Explain how to convert pipeline data to the JavaScript Object Notation (JSON) format.
- Explain how to convert pipeline data to the HTML format.
- Export data.
- Explain how to send pipeline data to other locations.

Writing output to a file

Windows PowerShell provides several ways to write output to a file. The **Out-File** command can accept input from the pipeline and write that data to a file. It can render objects as text by using the same technique that Windows PowerShell uses to render objects as text for on-screen display. That is, whatever you pipe into **Out-File** is the same as what would otherwise appear on the screen.

Windows PowerShell also supports converting and exporting objects, which the next lesson discusses. The **Out-File** behavior differs from converting or exporting the objects, because you don't change the form of the objects. Instead, Windows PowerShell captures what would have appeared on the screen.

Various **Out-File** parameters allow you to specify a file name, append content to an existing file, specify character encoding, and more.

Windows PowerShell also supports the text redirection operators (`>` and `>>`) that **cmd.exe** uses. These operators act as an alias for **Out-File**. The greater than sign (`>`) at the end of a pipeline directs output to a file, overwriting the content. Two consecutive greater than signs (`>>`) direct output to a file, appending the output to any text already in the file.

Out-File is the easiest way to move data from Windows PowerShell to external storage. However, the text files that **Out-File** creates are usually intended for viewing by a person. Therefore, it is frequently difficult or impractical to read the data back into Windows PowerShell in any way that allows the data to be manipulated, sorted, selected, and measured.

- **Out-File** writes whatever is in the pipeline to a text file
- The `>` and `>>` redirection operators are also supported
- The text file is formatted exactly the same as the data would have appeared on the screen—no conversion to another form occurs
- Unless the data has been converted to another form, the resulting text file is usually suitable for viewing only by a person
- As you start to build more complex commands, you need to keep track of what the pipeline contains at each step

Out-File does not produce any output of its own. This means that the command does not put objects into the pipeline. After you run the command, you should expect to see no output on the screen.

Keeping track of what the pipeline contains

As you begin to write more complex commands in Windows PowerShell, you need to become accustomed to keeping track of the pipeline's contents. As each command in the pipeline runs, the command might produce output different from the input it received. Therefore, the next command will work with something different. For example, look at the following example:

```
Get-Service |  
Sort-Object -Property Status, Name |  
Select-Object -Property DisplayName,Status |  
Out-File -FilePath ServiceList.csv
```

The preceding example contains five commands in a single command line, or pipeline:

- After **Get-Service** runs, the pipeline contains objects of type **System.ServiceProcess.ServiceController**. These objects have a known set of properties and methods.
- After **Sort-Object** runs, the pipeline still contains those **ServiceController** objects. **Sort-Object** produces output that has the same kind of object that was put into it.
- After **Select-Object** runs, the pipeline no longer contains **ServiceController** objects. Instead, it contains objects of type **Selected.System.ServiceProcess.ServiceController**. This behavior indicates that the objects derive from the regular **ServiceController** but have had some of their members removed. In this case, the objects contain only their **DisplayName** and **Status** properties, so you can no longer sort them by **Name**, because that property no longer exists.
- After **Out-File** runs, the pipeline contains nothing. Therefore, nothing appears on the screen after this complete command runs.

 **Note:** When you have a complex, multiple-command pipeline such as this one, you might have to debug it if it does not run correctly the first time. The best way to debug is to start with just one command and see what it produces. Then add the second command, and see what happens. Continue to add one command at a time, verifying that each one produces the output you expect before you add the next command. For example, the output of the previous command might not appear to be correctly sorted, because the sorting was done on the **Name** property and not the **DisplayName** property.

Converting output to CSV

Windows PowerShell includes the ability to convert pipeline objects to other forms of data representation. For example, you can convert a collection of objects to the CSV format.

Converting to CSV is useful for viewing and manipulating large amounts of data, because you can easily open the resulting file in a program such as Microsoft Excel.

• The commands are:

- **ConvertTo-CSV**
- **Export-CSV**

• The commands send:

- Properties as headers
- No type information

• You can easily open large CSV files in Excel

Windows PowerShell uses two distinct verbs for conversion: **ConvertTo** and **Export**. A command that uses **ConvertTo**, such as **ConvertTo-Csv** accepts objects as input from the pipeline and produces converted data as output to the pipeline. That is, the data remains in Windows PowerShell. You can pipe the data to another command that writes the data to a file or manipulates it in another way. Here is an example:

```
Get-Service | ConvertTo-Csv | Out-File Services.csv
```

A command that uses the **Export**, such as **Export-Csv**, performs two operations: it converts the data and then writes the data to external storage, such as a file on disk. Here is an example:

```
Get-Service | Export-Csv Services.csv
```

Export commands, such as **Export-Csv**, basically combine the functionality of **ConvertTo** with a command such as **Out-File**. Export commands do not usually put any output into the pipeline. Therefore, nothing appears on the screen after an export command runs.

A key part of both operations is that the form of the data changes. The structure referred to as *objects* no longer contains the data, which is instead represented in another form entirely. When you convert data to another form, it is generally more difficult to manipulate within Windows PowerShell. For example, you cannot easily sort, select, or measure data that has been converted.

As noted earlier, the output of **ExportTo-Csv** is a text file. The command writes the property names to the first row, as headers. One advantage of the CSV output format is that Windows PowerShell also supports importing the format via the **Import-Csv** command. **Import-Csv** creates objects that have properties matching the columns in the CSV file.

Converting output to XML

Windows PowerShell also supports writing output in the XML format. XML separates the data from the display format. The data then becomes highly portable and can be consumed by other processes and applications that do not need to know what format the data was originally presented in.

Another advantage of XML is that properties containing multiple values are easily identifiable, because a new *XML element*, which is a data container in the XML file, is created for each value. Finally, standard ways exist to query, parse, and transform XML data.

- **ConvertTo-CliXml**
- **Export-CliXml**
- Portable data format
- Multivalue properties become individual entries

Windows PowerShell converts data to XML by using the **ConvertTo-Clixml** and **Export-Clixml** commands. As with **ConvertTo-Csv** and **Export-Csv**, the **ConvertTo** version of the command does not send output to a file but leaves the output in memory for further processing. The **Export** version of the command creates an XML file in the full path specified by the *-Path* positional parameter.

Converting output to JSON

Another lightweight and increasingly popular data format is the JSON format. JSON is very popular in web application development because of its compact size and flexibility. As the name might suggest, it is very easy for JavaScript to process.

The JSON format represents data in name-value pairs that look and work like hash tables. So, like a hash table, JSON does not consider what kind of data exists in the name or value. It is up to the script or application consuming the JSON data to understand what kind of data is represented.

In Windows PowerShell, you create

JSON-formatted data by using the **ConvertTo-Json** command. As with the other **ConvertTo** commands, no output file is created. Unlike XML and CSV, however, JSON does not have an **Export** command for both converting the data and creating an output file. Therefore, you must use **Out-File** or one of the text redirection operators to send the JSON data to a file.

- The command is **ConvertTo-JSON**
- The advantages are:
 - Compactness
 - Ease of use, especially with JavaScript
 - A format like a hash table

Converting output to HTML

Sometimes, you need to display your Windows PowerShell output in a web browser or send it to a process, like the **Send-MailMessage** command, that accepts HTML input. Windows PowerShell supports this through the **ConvertTo-HTML** command. As with the other **ConvertTo** commands, no output file is created. You must direct the output by using **Out-File** or one of its aliases.

ConvertTo-HTML creates a simple list or table that is coded as HTML. You can control the HTML format in a limited fashion through a variety of parameters, such as:

- **-Head**. Specifies the content of an HTML **head** section.
- **-Title**. Sets the value of the HTML **title** tag.
- **-PreContent**. Defines any content that should appear before the table or list output.
- **-PostContent**. Defines any content that should appear after the table or list output.

- The command is **ConvertTo-HTML**
- The command creates a table or list in HTML
- You must pipe the output to a file
- The parameters include:
 - **-Head**
 - **-Title**
 - **-PreContent**
 - **-PostContent**

Demonstration: Exporting data

In this demonstration, you will see different ways to convert and export data.

Demonstration Steps

1. Convert a list of processes to HTML.
2. Create a file named **Procs.html** that contains an HTML-formatted list of processes.
3. Convert a list of services to CSV.
4. Create a file named **Serv.csv** that contains a CSV-formatted list of services.
5. Open **Serv.csv** in Notepad, and decide whether all the data was retained.

Additional output options

Windows PowerShell provides a variety of other options for controlling output. For example, you might not want to scroll through a long list of data returned by a command such as **Get-ADUser**.

You can use the **Out-Host** command to force Windows PowerShell to pause during each page of data it returns. You can force **Out-Host** to page the output when you use the *-Paging* parameter, which causes Windows PowerShell to prompt you for input after it displays one page of data on the screen.

- **Out-Host** allows more control of on-screen output
- **Out-Printer** sends output to a printer
- **Out-GridView** creates an interactive, spreadsheet-like view of the data

If you want to print output, you can use the **Out-Printer** command to pipe data to your default printer (if you specify no parameter) or to a specific printer (if you specify the *-Name* positional parameter). If you want to view, sort, filter, and analyze output, and you do not need or want to keep a more permanent copy of the data stored in a CSV file or spreadsheet, you can send the output to a grid view window by using the **Out-GridView** command. The grid view window is an interactive window that works very much like an Excel spreadsheet. It allows you to sort different columns, filter the data, and even copy data from the window to other programs. The one thing you cannot do is save the data directly from the grid view window.

Question: What other data formats might you want to convert data to or from?

Lab D: Sending output to a file

Scenario

You need to change some information about Active Directory users and create a report of those changes. You are evaluating data formats for reporting the information changes. You must convert data to different formats and decide which ones are the most appropriate for your needs.

Objectives

After completing this lab, you will be able to convert objects to different formats.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, select **10961C-LON-DC1**, and then in the **Actions** pane, select **Start**.
3. In the **Actions** pane, select **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Exercise 1: Converting objects

Scenario

In this exercise, you will write commands that query Active Directory users and make changes to information about them. You will also write commands that store the user data in various file formats. Then you will review the files to determine which data format you think is the most useful.

The main tasks for this exercise are as follows:

1. Update Active Directory user information.
 2. Produce an HTML report listing the Active Directory users in the IT department.
 3. Prepare for the next module.
- **Task 1: Update Active Directory user information**
1. Sign in to the **LON-CL1** as **Adatum\Administrator** with the password of **Pa55w.rd**.
 2. Start **Windows PowerShell** as an administrator.
 3. Display the name, department, and city for all the users in the IT department who are located in London, in alphabetical order by name.

4. Set the **Office** location for all the users to **LON-A/1000**.
5. Display the list of users again, including the office assignment for each user.

► **Task 2: Produce an HTML report listing the Active Directory users in the IT department**

1. View the help for **ConvertTo-Html**.
2. Display the same list again, and then convert the list to an HTML page. Store the HTML data in **E:\UserReport.html**. Have the word **Users** appear before the list of users.
3. Use Internet Explorer to view **UserReport.html**.
4. Display the same list again, and then convert it to XML.
5. Use Internet Explorer to view **UserReport.xml**.
6. Display a list of all the properties of all the Active Directory users in a CSV file.
7. Open the CSV file in Notepad.
8. Open the CSV file in Excel.

Results: After completing this exercise, you should have converted Active Directory user objects to different data formats.

► **Task 3: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Question: Can you use **ConvertTo-Csv** or **Export-Csv** to create a file delimited by a character other than a comma? For example, can you create a tab-delimited file?

Question: The HTML data produced by **ConvertTo-Html** looks very plain. The HTML standard offers a way to specify visual styles for an HTML document. This is known as *Cascading Style Sheets (CSS)*. Does the command offer a way to attach a style sheet?

Module Review and Takeaways

Best Practice

For the best performance, remember to move filtering actions as close to the beginning of the command line as possible. Sometimes, that means using a filtering capability of a regular command instead of using **Where-Object**.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The <code>\$_</code> variable does not work.	
The variable name <code>\$_</code> is confusing to read.	

Review Question

Question: The `$_` and `$PSItem` variable names were used several times in this module. Why might you decide to use one over the other?

Real-world Issues and Scenarios

One potentially challenging aspect of Windows PowerShell is that you can frequently achieve the same result in several ways. Different people select different techniques based on their experiences, but that does not necessarily make one technique better or worse than the others. Consider the following:

```
Get-Service | Select-Object -Property Name
Gsv | Select Name
Get-Service | ForEach Name
Get-Service | % { $_.Name }
Get-Service | ft name
```

In Windows PowerShell 3.0 or newer, the preceding five commands produce the same result: a list of service names. As you explore Windows PowerShell, and especially as you read examples written by other people or provided by your instructor, be aware that just one correct way to use Windows PowerShell does not exist. Part of using Windows PowerShell is being able to understand many approaches, arrangements of syntax, and techniques.

Studyjní materiály Okškolení

Studijní materiály Okškolení

Module 4

Understanding how the pipeline works

Contents:

Module Overview	4-1
Lesson 1: Passing pipeline data	4-2
Lesson 2: Advanced techniques for passing pipeline data	4-8
Lab: Working with pipeline parameter binding	4-13
Module Review and Takeaways	4-16

Module Overview

In this module, you will learn how the Windows PowerShell command-line interface passes objects from one command to another in the pipeline. Windows PowerShell has two techniques it can use: passing data **ByValue** and **ByPropertyName**. By knowing how these techniques work and which one to use in a given scenario, you can construct more useful and complex command lines.



Additional Reading: You can read more about the pipeline in Windows PowerShell at:
<https://aka.ms/raerme>

Objectives

After completing this module, you will be able to:

- Pass data by using the **ByValue** and **ByPropertyName** techniques.
- Describe the advanced techniques for passing pipeline data.

Lesson 1

Passing pipeline data

In this lesson, you will learn about the two ways that Windows PowerShell matches incoming pipeline data to the parameters of a cmdlet.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how commands receive input by using parameter binding.
- Identify parameters that can receive input by value.
- Explain how to pass pipeline data by using the **ByValue** technique.
- Pass data by using the **ByValue** technique.
- Explain how to pass pipeline data by using the **ByPropertyName** technique.
- Identify parameters that can accept input by property name.
- Pass data by using the **ByPropertyName** technique.

Pipeline parameter binding

A Windows PowerShell command can only use one of its parameters each time it runs to specify the object on which it operates. When piping data from one command to another, you do not need to specify that parameter. This makes the complete command statement easier to read. However, it might not be obvious how the command works when you run the command. Consider the following command:

```
Get-ADUser -Filter {Name -eq 'Perry Brill'}
| Set-ADUser -City Seattle
```

- Commands accept input only from one parameter

```
Get-ADUser -Filter {Name -eq 'Perry Brill'}
| Set-ADUser -City Seattle
```

- In the preceding example, two parameters are specified for **Set-ADUser**:
 - The one parameter you see is **-City**
 - Another parameter is used invisibly as part of *pipeline parameter binding*
- Two techniques for pipeline parameter binding:
 - **ByValue** is always tried first
 - **ByPropertyName** is tried if **ByValue** fails

In this example, **Set-ADUser** accepts input in an indirect manner; that is, it needs a user object as input. If running **Set-ADUser** directly, you might pass the user name to identify the user, but this command does not do this. Instead, the object that **Get-ADUser** produces is picked up out of the pipeline by **Set-ADUser**. **Set-ADUser** is passed two parameters, not the one parameter that you can see. Windows PowerShell invisibly uses the other parameter in a process that is known as *pipeline parameter binding*.

When you connect two commands in the pipeline, pipeline parameter binding takes the output of the first command and decides what to do with it. The process selects one of the parameters of the second command to receive that output. Windows PowerShell has two techniques that it uses to make that decision. The first technique—the one that Windows PowerShell always tries to use first—is named **ByValue**. The second technique is named **ByPropertyName**, and it is used only when **ByValue** fails.

The ability to accept pipeline output is part of the definition of the parameter within the cmdlet code.

Identifying ByValue parameters

If you read the full Help for a command, you can see the pipeline input capability of each parameter. For example, in the Help file for **Sort-Object**, you will find the following information:

```
-InputObject <PSObject>
    Specifies the objects to be sorted.
    To sort objects, pipe them to Sort-Object.
    Required?           false
    Position?          Named
    Default value      None
    Accept pipeline input? true
    (ByValue)
    Accept wildcard characters? false
```

The Help file for a command indicates the parameters that can accept the pipeline input **ByValue**

Required?	false
Position?	Named
Default value	None
Accept pipeline input?	true (ByValue)
Accept wildcard characters?	false

The **Accept pipeline input?** attribute is **true** because the **-InputObject** parameter accepts pipeline input. Additionally, Help shows a list of techniques the parameter supports. In this case, it supports only the **ByValue** technique.

Passing data by using ByValue

When passing data by using **ByValue**, a parameter can accept complete objects from the pipeline when those objects are of the type that the parameter accepts. A single command can have more than one parameter accepting pipeline input **ByValue**, but each parameter must accept a different kind of object.

For example, **Get-Service** can accept pipeline input **ByValue** on both its **-InputObject** and **-Name** parameters. Each of those parameters accept a different kind of object. **-InputObject** accepts objects of the type **ServiceController**, and **-Name** accepts objects of the type **String**. Consider the following example:

```
'BITS','WinRM' | Get-Service
```

String objects in the pipeline
"BITS","WinRM" | Get-Service -Name
Attaches invisibly to the parameter that accepts String objects from the pipeline ByValue

Here, two string objects are piped into **Get-Service**. They attach to the **-Name** parameter because that parameter accepts that kind of object, **ByValue**, from the pipeline.

To predict the function that Windows PowerShell performs with the object in the pipeline, you need to determine the kind of object in the pipeline. For this purpose, you can pipe the object to **Get-Member**. The first line of output tells you the kind of object that the pipeline contained. For example:

```
PS C:\> "BITS","WinRM" | Get-Member
 TypeName: System.String
 Name          MemberType
 ----          -----

```

Here, the pipeline contains objects of the type **System.String**. Windows PowerShell often abbreviates type names to include only the last portion. In this example, that is **String**.

Then you examine the full Help for the next command in the pipeline. In this example, it is **Get-Service**, and you would find that both the **-InputObject** and **-Name** parameters accept input from the pipeline **ByValue**. Because the pipeline contains objects of the type **String**, and because the **-Name** parameter accepts objects of the type **String** from the pipeline **ByValue**, the objects in the pipeline attach to the **-Name** parameter.

Generic object types

Windows PowerShell recognizes two generic kinds of object, **Object** and **PSObject**. Parameters that accept these kinds of objects can accept any kind of object. When you perform **ByValue** pipeline parameter binding, Windows PowerShell first looks for the most specific object type possible. If the pipeline contains a **String**, and a parameter can accept **String**, that parameter will receive the objects.

If there is no match for a specific data type, Windows PowerShell will try to match generic data types. That behavior is why commands like **Sort-Object** and **Select-Object** work. Each of those commands have a parameter named **-InputObject** that accepts objects of the type **PSObject** from the pipeline **ByValue**. This is why you can pipe any type of object to those commands. Their **-InputObject** parameter will receive any object from the pipeline because it accepts objects of any kind.

Demonstration: Passing data by using ByValue

In this demonstration, you will see how Windows PowerShell performs pipeline parameter binding **ByValue**.

Demonstration Steps

1. On **LON-CL1**, in Windows PowerShell ISE, type the following command:

```
Get-Service -Name BITS | Stop-Service
```

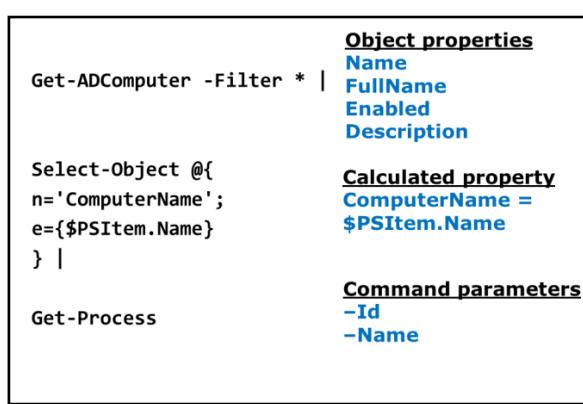
2. Discover what kind of object the first command in step 1 produces.
3. Discover what parameters of the second command in step 1 can accept pipeline input **ByValue**.
4. Decide which parameter of the second command in step 1 will receive the output of the first command.

Passing pipeline data ByPropertyName

If Windows PowerShell is unable to bind pipeline input by using the **ByValue** technique, it tries to use the **ByPropertyName** technique. When using the **ByPropertyName** technique, Windows

PowerShell attempts to match a property of the object passed to a parameter of the command to which the object was passed. This match occurs in a simple manner. If the input object has a **Name** property, it will be matched with the parameter **Name** because they are spelled the same.

However, it will only pass the property if the parameter is programmed to accept a value by property name. This means that you can pass output from one command to another when they do not logically go together.



For example:

```
Get-LocalUser | Stop-Process
```

The first command puts objects of the type **LocalUser** into the pipeline. The second command has no parameters that can accept that kind of object. The second command also has no parameters that accept a generic **Object** or **PSObject**. Therefore, the **ByValue** technique fails.

Because the **ByValue** technique fails, Windows PowerShell changes to the **ByPropertyName** technique. To predict what it will try to do, you can review the properties of the objects that the first command produces. In this example, run the following command:

```
Get-LocalUser | Get-Member
```

You also need to make a list of parameters of the second command that can accept pipeline input by using **ByPropertyName**. To make that list, view Help for the second command:

```
Get-Help Stop-Process -ShowWindow
```

By doing this, you will see that the **Stop-Process** command has more than one parameter that accepts pipeline input by using **ByPropertyName**. Those parameters are **-Name** and **-Id**. The objects that **Get-LocalUser** produces do not have an **ID** property, so the **-Id** parameter is not considered. The objects that **Get-LocalUser** produces have a **Name** property. Therefore, the contents of the **Name** property attach to the **-Name** parameter of **Stop-Service**. This means that **Stop-Service** will try and stop a service with a name that is the same as a user. Although you would not want to do this in a real-world scenario, if you try this step in Windows PowerShell, you will notice that any errors that you receive are because a process cannot be found with the target name.

Renaming properties

Most often, a property name from an output object does not match the name of an input parameter exactly. You can change the name of the property by using **Select-Object** and create a calculated property. For example, to view the processes running on all computers in your Windows Server Active Directory, try running the following command:

```
Get-ADComputer -Filter * | Get-Process
```

However, this does not work. No parameter for **Get-Process** matches a property name for the output of **Get-ADComputer**. View the output of **Get-ADComputer | Get-Member** and **Get-Help Get-Process** and you will see that what you want is to match the **Name** property of **Get-ADComputer** with the **-ComputerName** parameter of **Get-Process**. You can do that by using **Select-Object** and changing the property name for the **Get-ADComputer** command's **Name** property to **ComputerName**, and then passing the results to **Get-Process**. The following command will work:

```
Get-ADComputer -Filter * | Select-Object @{n='ComputerName';e={$PSItem.Name}} | Get-Process
```

Another common use of the **ByPropertyName** technique is when you import data from comma-separated value (CSV) or similar files, and you feed that data to a command so that you can process a specific list of users, computers, or other resources. You will learn how to import data in Module 8, "Basic scripting."

Identifying ByPropertyName parameters

Like **ByValue** parameters, you can see the parameters that accept pipeline input by using the **ByPropertyName** technique and examining the full Help for the command. For example, run the command **Get-Help Stop-Process -Full** and you will see the following parameters:

```
-ID <Int32[]>
    Required?          true
    Position?         0
    Default value     None
    Accept pipeline input? True
    (ByPropertyName)
        Accept wildcard characters? False
-InputObject <Process[]>
    Required?          true
    Position?         0
    Default value     None
    Accept pipeline input? True (ByValue)
    Accept wildcard characters? false
-Name <String[]>
    Required?          true
    Position?         named
    Default value     None
    Accept pipeline input? True (ByPropertyName)
    Accept wildcard characters? false
```

The full Help for a command lists the parameters that accept pipeline input by using **ByPropertyName**

Required?	false
Position?	named
Default value	Local computer
Accept pipeline input?	true (ByPropertyName)
Accept wildcard characters?	false

Notice that two parameters potentially accept input **ByPropertyName**. Keep in mind that only one parameter can accept input at a time. Further, because there is a parameter that accepts **ByValue**, Windows PowerShell will try that one first.

It is also possible to have a single parameter that accepts pipeline input by using both **ByValue** and **ByPropertyName**. Again, Windows PowerShell will always try **ByValue** first and will use **ByPropertyName** only if **ByValue** fails.

Demonstration: Passing data ByPropertyName

In this demonstration, you will see how to use **ByPropertyName** parameters.

Demonstration Steps

1. On **LON-CL1**, try to view a list of all processes that are running on all computers listed in Active Directory by using the following command:

```
Get-ADComputer LON-DC1 | Get-Process
```

2. Review the error.
3. View the properties of the object that **Get-ADComputer** returns.
4. View the parameters for **Get-Process**, and then identify any parameters that might work with properties of objects that **Get-ADComputer** passes.
5. Attempt to view the list of processes again by using a calculated property to pass appropriate input to **Get-Process**.

Question: Why do most commands that use the noun **Object** have an **-InputObject** parameter that accepts objects of the type **Object** or **PSObject**?

Lesson 2

Advanced techniques for passing pipeline data

In this lesson, you will learn about additional advanced techniques for passing pipeline data, such as overriding parameters, using parenthetical commands, and expanding property values. You will learn the syntax for implementing these techniques and scenarios where these techniques are valuable.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use manual parameters to override the pipeline.
- Override the pipeline.
- Explain how to specify input by using parenthetical commands instead of the pipeline.
- Use parenthetical commands.
- Explain how to expand object properties into simple values.
- Expand property values.

Using manual parameters to override the pipeline

Any time that you manually type a parameter for a command, you override any pipeline input that the parameter might have accepted. You do not force Windows PowerShell to select another parameter for pipeline parameter binding.

Consider the following example:

```
Get-Process -Name Notepad | Stop-Process -Name Notepad
```

In this example, **Get-Process** gets a process object that has the **Name** property and passes that object to **Stop-Process**. However, in this example, the **-Name** parameter was already used manually. That stops pipeline parameter binding. Windows PowerShell will not look for another parameter to bind the input even though there are parameters that accept other properties. In the example above, a parameter that Windows PowerShell wanted to use is taken, so the process is over.

In this case, and in most cases, you will receive an error even though the property value matches the value you specified for the parameter. For the above command, you receive the following error:

```
Stop-Process : The input object cannot be bound to any parameters for the command either because the command does not take pipeline input or the input and its properties do not match any of the parameters that take pipeline input.
```

The error is misleading. It says, "... the command does not take pipeline input." However, the command does take pipeline input. In this example, you have disabled the command's ability to accept the pipeline input because you manually specified the parameter that Windows PowerShell wanted to use.

Demonstration: Overriding the pipeline

In this demonstration, you will see an example of an error when you manually specify a parameter that Windows PowerShell would usually have used in pipeline parameter binding.

Demonstration Steps

1. Open Windows PowerShell as an administrator.
2. Open **Notepad.exe**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-Process -Name Notepad | Stop-Process
```
4. Open Notepad.exe again.
5. In the console, type the following command, and then press Enter:

```
Get-Process -Name Notepad | Stop-Process -Name Notepad
```

This command generates an error.

6. Review the error, and then discuss how using the **-Name** parameter causes an error.

Using parenthetical commands

Another option for passing the results of one command to the parameters of another is by using parenthetical commands. A *parenthetical command* is a command that is enclosed in parentheses. Just as in math, parentheses tell Windows PowerShell to execute the enclosed command first. The parenthetical command runs, and the results of the command are inserted in its place.

You can use parenthetical commands to pass values to parameters that do not accept pipeline input. This means you can have a pipeline that includes data inputs from multiple sources. Consider the following command:

```
Get-ADGroup "London Users" | Add-ADGroupMember -Members (Get-ADUser -Filter {City -eq 'London'})
```

- Any command or commands in parentheses will be run first
- The results will be inserted in place of the parenthetical command
- Works with any parameter if the command produces the kind of object that the parameter expects

```
Get-ADGroup "London Users" | Add-ADGroupMember -Members (Get-ADUser -Filter {City -eq 'London'})
```

In this example, output of **Get-ADGroup** passes to **Add-ADGroupMember**, telling it which group to modify. However, that is only part of the information needed. We also need to tell **Add-ADGroupMember** what users to add to the group. The **-Members** parameter does not accept piped input, and even if it did, we have already piped data to the command. Therefore, we need to use a parenthetical command to provide a list of users that we want added to the group.

Parenthetical commands do not rely on pipeline parameter binding. They work with any parameter if the parenthetical command produces the kind of object that the parameter expects.

 **Note:** A common scenario for using parenthetical commands is to read input from a file and pass that data to a parameter. You will learn more about reading data from files in Module 8, "Basic scripting."

Demonstration: Using parenthetical commands

In this demonstration, you will see how to use parenthetical commands.

Demonstration Steps

1. On **LON-CL1**, open Windows PowerShell as an administrator.
2. Create an Active Directory global security group named **London Users**.
3. In the console, type the following command, and then press Enter:

```
Get-ADGroup "London Users" | Add-ADGroupMember
```

Note that a prompt to enter users appears. Press Enter. You will see an error because no member was specified.

4. In the console, type the following command, and then press Enter:

```
Get-ADGroup "London Users" | Add-ADGroupMember -Members (Get-ADUser -Filter {City -eq 'London'})
```

5. Retrieve a list of members belonging to the security group **London Users** and confirm that new members were added.

Expanding property values

You can use parenthetical commands to provide parameter input without using the pipeline. In some cases, however, you might have to manipulate the objects produced by a parenthetical command so that the command's output is of the type that the parameter requires.

For example, you might want to list all the processes that are running on every computer in the domain. In this example, imagine that you have a very small lab domain that contains just a few computers. You can get a list of every computer in the domain by running the following command:

```
Get-ADComputer -Filter *
```

- The **-ExpandProperty** parameter of **Select-Object** expands, or extracts, the contents of a single property
- Instead of returning an object with many properties, the command returns a simpler value:
 - Converts multiple value properties to a series of single items
 - Additionally, works when piping commands

However, this command produces objects of the type **ADComputer**. You could not use those objects directly in a parenthetical command such as the following:

```
Get-Process -ComputerName (Get-ADComputer -Filter *)
```

The **-ComputerName** parameter expects objects of the type **String**. However, the parenthetical command does not produce **String** type objects. The **-ComputerName** parameter only wants a computer name. However, the command provides it an object that contains a name, an operating system version, and several other properties.

You could try the following command:

```
Get-Process -ComputerName (Get-ADComputer -Filter * | Select-Object -Property Name)
```

This command selects only the **Name** property. This property is still a member of a whole **ADComputer** object. It is the **Name** property of an object. Although the **Name** property contains a string, it is not itself a string. The **-ComputerName** parameter expects a string, not an object with a property. Therefore, that command does not work either.

The following command achieves the goal of passing the computer name as a string to the **-ComputerName** parameter:

```
Get-Process -ComputerName (Get-ADComputer -Filter * | Select-Object -ExpandProperty Name)
```

The **-ExpandProperty** parameter accepts one, and only one, property name. When you use that parameter, only the contents of the specified property are produced by **Select-Object**. Some people refer to this as *extracting the property contents*. The official description of the feature is *expanding the property contents*.

In the preceding command, the result of the parenthetical command is a collection of strings that are passed as individual strings, not an array, and that is what the **-ComputerName** parameter expects. The command will work correctly; of course, it might produce an error if one or more of the computers cannot be reached on the network.

Expanding property values also works when piping output. Consider the following example:

```
Get-ADUser Ty -Properties MemberOf | Get-ADGroup
```

This command returns an error because Windows PowerShell cannot match the **MemberOf** property to any property of **Get-ADGroup**.

However, if you expand the value of the **MemberOf** property, as in the following example, Windows PowerShell can match the resulting output to a value that **Get-ADGroup** understands as valid input:

```
Get-ADUser Ty -Properties MemberOf | Select-Object -ExpandProperty MemberOf | Get-ADGroup
```

Demonstration: Expanding property values

In this demonstration, you will see how to use parameter expansion to provide input from a parenthetical command.

Demonstration Steps

1. Open the Windows PowerShell Integrated Scripting Environment (ISE).
2. Run a command that will list all computers in the domain.
3. Run a command that uses a parenthetical command to display a list of services from every computer in the domain.
4. Run a command that shows the kind of object that is produced when you retrieve information about every computer account in the domain.
5. Review the Help for **Get-Service** to see what kind of object its **-ComputerName** parameter expects.
6. Run a command that selects only the **Name** property of every computer in the domain.
7. Run a command that shows the kind of object that the previous command produced.
8. Run a command that extracts the contents of the **Name** property of every computer in the domain.
9. Run a command that shows the kind of object that the previous command produced.
10. Modify the command in step 3 to use the command in step 8 as the parenthetical command.
11. Run the command that you created in step 10.

Question: What are some reasons you would use parenthetical commands instead of the pipeline?

Lab: Working with pipeline parameter binding

Scenario

You are creating and troubleshooting Windows PowerShell commands. You must predict and control how Windows PowerShell will pass data from one command to another so that the commands run correctly.

Objectives

After completing this lab, you will be able to predict the behavior of commands in the pipeline.

Lab Setup

Estimated Time: **45 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

The changes that you make during this lab will be lost if you revert your VMs at another time during class.

For this lab, you will use the available VM environment. Before you begin the lab, you must follow these steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the VM starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Perform this lab in the Window PowerShell console on the **10961C-LON-CL1** VM.

Exercise: Predicting pipeline behavior

Scenario

You must review several Windows PowerShell commands and determine whether they will work. Some commands use pipeline input, but other commands do not. Without running the commands in their entirety, you must decide whether they will achieve the stated goal that the lab task describes.

You also must write several Windows PowerShell commands that will achieve goals stated in the lab task. You must not run these commands in Windows PowerShell. Instead, write them on paper.

The main tasks for this exercise are as follows:

1. Review existing commands.
2. Write new commands that perform the specified tasks.
3. Prepare for the next module.

Studijní materiály Okškolení

► Task 1: Review existing commands

 **Note:** For these tasks, you may run individual commands and **Get-Member** to see what kinds of objects the commands produce. You may also view the Help for any of these commands. However, do not run the whole command as shown. If you do run the whole command, it might produce an error. The error does not mean the command is written incorrectly.

1. This command's intent is to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Get-Service -Name *
```

Will the command achieve the goal?

2. This command's intent is to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-Service -Name *
```

Will the command achieve the goal?

3. This command's intent is to query an object from every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-WmiObject -Class Win32_BIOS
```

Will the command achieve the goal?

4. This command's intent is to list the services that are running on every computer in the domain:

```
Get-Service -ComputerName (Get-ADComputer -Filter *)
```

Will the command achieve the goal?

5. This command's intent is to list the Security event log entries from the server **LON-DC1**:

```
Get-EventLog -LogName Security -ComputerName (Get-ADComputer -LON-DC1 | Select -ExpandProperty Name)
```

Will the command achieve the goal?

► Task 2: Write new commands that perform the specified tasks

 **Note:** In each of these tasks, you are to write a command that achieves a specified goal. *Do not run these commands.* Write them on paper.

You may run individual commands and pipe their output to **Get-Member** to see what objects those commands produce. You may also read the Help for any command.

1. Write a command that uses **Get-EventLog** to display the most recent 50 System event log entries from each computer in the domain.

2. Write a command that uses **Set-Service** to set the start type of the **WinRM** service to **Auto** on every computer in the domain. Do not use a parenthetical command.

Results: After completing this exercise, you will have successfully reviewed and written several commands in the Windows PowerShell command-line interface.

► Task 3: Prepare for the next module

When you finish the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: Why do some commands accept pipeline input for a parameter such as **-ComputerName**, but other commands do not?

Question: Do you ever have to rely on pipeline input? Could you just rely on parenthetical commands?

Studijní materiály Okškolení

Module Review and Takeaways

Review Question

Question: Because Windows PowerShell handles pipeline input binding invisibly, it can be difficult to troubleshoot. Are there any tools that can help you troubleshoot pipeline input?

Best Practices

It is easy to start using Windows PowerShell and not think about what it is doing for you. Always take a moment to examine each command that you write, and think about what Windows PowerShell will do. Think about what objects each command will produce and how those will pass to the next command.

Real-world Issues and Scenarios

Sometimes, command authors do not realize how useful and important pipeline input can be, and they do not create their parameters to accept pipeline input. All that you can do in those cases is to submit a request to the command author to support pipeline input in a future release.

Module 5

Using PSProviders and PSDrives

Contents:

Module Overview	5-1
Lesson 1: Using PSProviders	5-2
Lesson 2: Using PSDrives	5-5
Lab: Using PSProviders and PSDrives	5-14
Module Review and Takeaways	5-18

Module Overview

A **PSProvider** is a Windows PowerShell adapter that makes some form of storage resemble a hard drive. A **PSDrive** is an actual connection to a form of storage. These two technologies let you work with many forms of storage by using the same commands and techniques that you use to manage the file system. In this module, you will learn to work with **PSProviders** and **PSDrives**.



Additional Reading: For more information, refer to: "Managing Windows PowerShell Drives" at: <https://aka.ms/v25ssg>

Objectives

After completing this module, you will be able to:

- Use **PSProviders**.
- Use **PSDrives**.

Lesson 1

Using PSProviders

In this lesson, you will learn about **PSProviders**, which are adapters that connect Windows PowerShell to data stores. Providers give you an easy-to-understand and consistent interface for working with these data stores. This makes learning to work with the providers easier, and it also allows you to reuse scripts as you change the underlying technologies with which you are working.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of **PSProviders**.
- Compare different **PSProvider** capabilities.
- Explain how to view **PSProvider** help files.
- Explain how to view a list of providers and help for a specific provider.

What are Windows PowerShell providers?

As you learned in the Module Overview, a **PSProvider**, or just *provider*, is an adapter that makes some data stores resemble hard drives within Windows PowerShell. Because most administrators are already familiar with command-line management of hard drives, **PSProviders** help those administrators manage other forms of data storage by using familiar commands.

A provider presents data as a hierarchical store. Items such as folders can have sub-items that appear as subfolders. Items can also have properties, and providers let you manipulate both items and item properties by using a specific set of commands.

Managing a technology by using a provider is more difficult than managing it by using technology-specific commands. Individual commands perform specific actions, and the command name describes what the command does. For example, in Microsoft Internet Information Services (IIS), the **Get-WebSite** command retrieves IIS sites. When you use the IIS provider, you run the **Get-ChildItem IIS:\Sites** command instead.

The advantage of a **PSProvider** is that it is dynamic. For example, you might not know in advance what hard drives, folders, and files are installed and created on a computer. The **FileSystem** provider can dynamically adapt to whatever is on each computer. IIS is also managed, in part, by using a **PSProvider**. This provider can adapt to the Microsoft and third-party add-ins that are installed in IIS. Microsoft cannot easily write commands to manage everything in IIS, because new add-ins are created constantly. So even though management by using a provider is more complex, it is a better strategy when dealing with dynamic and extensible technologies.

- Adapt data stores to look like hard drives inside Windows PowerShell
- Allow management by using familiar file system management commands
- Good solution for dynamic or extensible technologies where all manageable components cannot be known in advance
- More complex than managing by using technology-specific commands

Different provider capabilities

The generic commands that you use to work with providers offer a superset of every feature that a provider might support. For example, the **Get-ChildItem** command includes the `-UseTransaction` parameter. However, the only built-in provider that supports the **Transactions** capability is the **Registry** provider. If you try to use the `-UseTransaction` parameter in any other provider, you will receive an error message. You will receive an error message whenever you use a common parameter that the provider does not support.

Run the **Get-PSPProvider** cmdlet to list the capabilities of each provider that loads into Windows PowerShell. The capabilities of each provider will be different because each provider connects to a different underlying technology.

Some important capabilities include:

- **ShouldProcess**. This is for providers that can support the `-WhatIf` and `-Confirm` parameters.
- **Filter**. This is for providers that support filtering.
- **Include**. This is for providers that can include items in the data store based on the name. Supports using wildcards.
- **Exclude**. This is for providers that can exclude items in the data store based on the name. Supports using wildcards.
- **ExpandWildcards**. This is for providers that support wildcards in their paths.
- **Credentials**. This is for providers that support alternative credentials.
- **Transactions**. This is for providers that support transacted operations.

You should always review the capabilities of a provider before you work with it. This helps to avoid unexpected errors when you try to use unsupported capabilities.

- Each provider can support one or more capabilities
- If a provider does not support a capability, you cannot use the corresponding parameters. For example:
 - Because the **Registry** provider is the only provider that supports the **Transactions** capability, only this provider can accept the `-UseTransaction` parameter

Accessing provider help

You can display a list of available providers by using the **Get-PSPProvider** cmdlet. Be aware that providers can be added into Windows PowerShell when you load modules, and they will not display until loaded. For example, running the **Import-Module ActiveDirectory** cmdlet or module autoloading when running an Active Directory cmdlet will load the **ActiveDirectory** module, which includes a **PSPProvider**.

- Run the **Get-PSPProvider** cmdlet for a list of providers
- Run the **Get-Help <provider-name>** cmdlet for provider-specific help
- Provider-specific help can offer better descriptions and examples than the help for generic commands

When you know the name of a provider, you can view its help if it is provided. For example, run the **Get-Help FileSystem** cmdlet to display help for the **FileSystem** provider. Provider help files frequently contain descriptions and examples that are specific to the data store to which the provider connects. You can also use the **-Category** parameter of the **Get-Help** cmdlet with **Provider** as the value to make sure that you are targeting the provider's help.

Commands that work with providers use the nouns **Item** and **ItemProperty**. Examples for every scenario might not be in commands' help, because the commands are designed to work with any provider. The intent of provider help is to supplement command help with more specific descriptions and examples.

Demonstration: Viewing PSProvider help

In this demonstration, you will see how to view a list of providers and help for a provider.

Demonstration Steps

1. Display a list of providers. Notice the capabilities listed for each one.
2. Load the **ActiveDirectory** module.
3. Display the list of providers again. Notice the new provider that the **ActiveDirectory** module added.
4. Display help for the **Registry** provider.

Question: What other kinds of **PSProviders** might exist as add-ins in the Windows PowerShell command-line interface?

Lesson 2

Using PSDrives

In this lesson, you will learn how to work with **PSDrives**. A **PSDrive** represents a specific form of storage that connects to Windows PowerShell by using a **PSProvider**. You need to understand how to work with **PSDrives** to use **PSProviders** successfully. In some cases, such as the file system, **PSDrives** are the primary interface for working with the underlying data.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and use of **PSDrives**.
- Identify the cmdlets for using **PSDrives**.
- Explain how to find, delete, and create files and directories.
- Explain how to use Windows PowerShell to manage the file system.
- Explain how to work with the registry.
- Explain how to use Windows PowerShell to manage the registry.
- Explain how to work with certificates.
- Explain how to work with other **PSDrives**.

What are PSDrives?

A **PSDrive**, or *drive*, is a connection to a data store. Each **PSDrive** uses a single **PSProvider** to connect to a data store. The **PSDrive** has all the capabilities of the **PSProvider** that is used to make the connection.

Names identify drives in Windows PowerShell, not only by single letters, although a name can consist of just one letter. For example, the drive **HKCU** connects to the **HKEY_CURRENT_USER** registry hive. Single-letter drive names typically connect to a **FileSystem** drive. For example, drive **C** connects to the physical drive **C** of a computer.

To create a new connection, use the **New-PSDrive** cmdlet. You must specify a unique drive name, the root location for the new drive, and the **PSProvider** that will make the connection. Depending on the capabilities of the **PSProvider**, you might also specify alternative credentials and other options.

Windows PowerShell always starts a new session with the existing drives:

- Registry drives **HKLM** and **HKCU**
- Local hard drives such as drive **C**
- Windows PowerShell storage drives **Variable**, **Function**, and **Alias**

- A drive represents a connected data store
- Drives use a **PSProvider** to connect to the store
- Drives have a name such as **C** or **Alias**
- Drive names do not include a colon (:)
- Drive paths do include a colon, for example **C:**
- Run the **New-PSDrive** cmdlet to map a new drive
- Run the **Get-PSDrive** cmdlet for a list of drives
- Windows PowerShell always starts with the same drives mapped

Studijní materiály Okškolení

- Web Services for Management (WS-Management) settings drive **WSMan**
- Environment variables drive **Env**
- Certificate store drive **CERT**

You can see a list of drives by running the **Get-PSDrive** cmdlet.

 **Note:** Drive names do not include a colon. Drive name examples include **Variable** and **Alias**. However, when you want to refer to a drive as a path, include a colon. For example, **Variable:** refers to the path to the **Variable** drive, just as **C:** refers to the path to drive **C**. Commands such as **New-PSDrive** require a drive name. Do not include a colon in the drive name when you use those commands.

Cmdlets for using PSDrives

Because Windows PowerShell creates **PSDrives** for local drives such as drive **C**, you might already be using some of the cmdlets that are associated with **PSDrives** without realizing it. **PSDrives** contain items, and items contain child items or item properties. The Windows PowerShell cmdlet names that work with **PSDrive** objects use the nouns **Item**, **ChildItem**, and **ItemProperty**.

You can use the **Get-Command** command with the **-Noun** parameter to view a list of commands that work on each **PSDrive** object and **Get-Help** to view the help for each command.

The following list describes the verbs that are associated with common **PSDrive** cmdlets:

Verb	Description
New	Creates a new item or item property
Set	Sets the value of an item or item property
Get	Displays properties of an item, child item, or value of an item property
Clear	Clears the value of an item or item property
Copy	Copies an item or item property from one location to another location
Move	Moves an item or item property from one location to another location
Remove	Deletes an item or item property
Rename	Renames an item or item property
Invoke	Performs the default action that is associated with an item

- **PSDrives** contain items, child items, and item properties
- Run the **Get-Command -Noun Item** (or **ChildItem** or **ItemProperty**) cmdlet for a list of commands that manage **PSDrive** contents
- Run the **Get-Command -Noun Location** cmdlet to find commands for managing **PSDrive** locations

The items in the various **PSDrives** behave differently. Although these commands work in all **PSDrives**, how the verbs act on the items in each **PSDrive** might vary. Additionally, other commands might work with those items. The other topics in this lesson describe how to work with specific **PSDrives**.

When you use commands that have the **Item**, **ChildItem**, and **ItemProperty** nouns, you typically specify a path to tell the command what item or items that you want to manipulate. Most of these commands have two parameters for paths:

- **-Path**. This typically interprets the asterisk (*) and the question mark (?) as wildcard characters. In other words, the path ***.txt** refers to all files ending in ".txt." This approach works correctly in the file system because the file system does not allow item names to contain the * or ? characters.
- **-LiteralPath**. This treats all characters as literals and does not interpret any character as a wildcard. The literal path ***.txt** means the item named "*.txt." This approach is useful in drives where the * and ? characters are allowed in item names, such as in the registry.

Working with PSDrive locations

In addition to the commands for working with **PSDrive** items and item properties, there are also commands for working with **PSDrive** working locations. *Working locations* are paths within **PSDrives** to items that can have child items, such as a file system folder or registry path.

The commands that manage **PSDrive** locations use the **Location** noun and include.

Verb	Description
Get-Location	Displays the current working location
Set-Location	Sets the current working location
Push-Location	Adds a location to the top of a location stack
Pop-Location	Changes the current location to the location at the top of a location stack

 **Note:** The **Push-Location** and **Pop-Location** cmdlets are the equivalent of the **pushd** and **popd** commands in the **cmd.exe** console. **Pushd** and **popd** are also aliases for those cmdlets. For more information about location stacks, refer to: "Push-Location" at: <https://aka.ms/idat4p>

Working with the file system

Administrators who are familiar with using the **cmd.exe** console most likely know commands to manage a file system. Common **cmd.exe** commands include **Dir**, **Move**, **Ren**, **RmDir**, **Del**, **Copy**, **MkDir**, and **Cd**. In Windows PowerShell, these common commands are provided as aliases, or functions, that map to equivalent **PSDrive** cmdlets.

Use the **Get-Alias** or **Get-Command** command to identify the cmdlets that map to these aliases and functions.

- **New-Item** creates files and folders:
 - Alternate commands: **mkdir**, **md**, **ni**
 - Requires the **-Path** and **-ItemType** parameters
- **Remove-Item** deletes files and folders:
 - Alternate commands: **del**, **erase**, **rd**, **ri**, **rm**, **rmdir**
 - Use **-Recurse** to delete files when deleting folders
- **Get-Item** and **Get-ChildItem** retrieve files and folders:
 - Alternate commands: **gi**, **gci**, **dir**, **ls**
 - Supports **-Exclude**, **-Include**, and **-Filter**

Keep in mind that the aliases and functions are not exact duplicates of the original **cmd.exe** commands. Their syntax is not the same as the original command; it is the syntax for the cmdlet. For example, instead of running **Dir /s** to obtain a directory listing that includes subdirectories, you run the **Get-ChildItem -Recurse** command. The parameters are the same whether you decide to use the cmdlet name or the alias. That means that you can run the command **Dir -Recurse**, but not **Dir /s**.

 **Note:** Because Windows PowerShell accepts a forward slash (/) or backslash (\) as a path separator, Windows PowerShell interprets **Dir /s** to display a directory listing of the folder named **s**. If a folder named **s** exists, the command will appear to work and will not display an error. If no such folder exists, it displays an error.

Create new files or folders

You can create new files and folders by using the **New-Item** command. You include the *-Path* parameter to define the name and location and the *-ItemType* parameter to specify if it is a file or directory that you wish to create.

Delete files or folders

You can remove files or folders with the **Remove-Item** command and the positional *-Path* parameter. To delete folders that contain files, you need to include the **-Recurse** switch so that the child file items are also deleted, or you will be asked to confirm the action.

Find and enumerate files or folders

Use the **Get-Item** command and the *-Path* parameter to retrieve a single file or folder. You can also retrieve the children of an item by including the * wildcard in the path. For example, the **Get-Item *** command will return all files and folders in the current directory. The **Get-Item *** command is equivalent to the **Get-ChildItem** command, which returns all the children of a specified path.

You can use the **Get-ChildItem** command with the **-Recurse** switch to enumerate through child files and folders.

The **FileSystem** provider also supports the *-Exclude*, *-Include*, and *-Filter* parameters, which modify the value of the *-Path* parameter and specify file and folder names to include or exclude in the retrieval process.

Demonstration: Managing the file system

In this demonstration, you will see how to manage the file system by using Windows PowerShell.

Demonstration Steps

1. Go to **C:** by using an alias that maps to a **cmd.exe** command.
2. Go to **C:\Windows** by using a cmdlet.
3. Map a new temporary drive named **WINDIR** to the **C:\Windows** folder.
4. Display a directory listing of the **WINDIR** drive by using an alias.
5. Display the directory listing for the **WINDIR** drive again by using a cmdlet.
6. Create a folder named **Temp** in the **E:\Mod05** folder.

Working with the registry

Experienced system administrators are familiar with running the **regedit** command to start the Registry Editor so that they can manage registry entries. You can also manage the registry through the **Registry** provider by using Windows PowerShell.

Windows PowerShell uses the **Registry** provider to create two **PSDrives** automatically, **HKLM** and **HKCU**, which represent the **HKEY_Local_Machine** and **HKEY_Local_User** registry hives, or subtrees. It is possible to create **PSDrives** for other hives by using the **New-PSDrive** command.

Registry keys are accessed by using the **Item** and **ChildItem** nouns, while values are accessed by using the **ItemProperty** noun. This is because Windows PowerShell considers values as properties of the key item.

For example, to return all the registry keys under the **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion** path, run the following command:

```
Get-ChildItem HKM:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

To return the registry values under the **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run** path, run the following command instead:

```
Get-ItemProperty HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Get-ItemProperty returns all the keys and their values under the specified path.

In Windows PowerShell 5.0 and newer, you can use the **Get-ItemPropertyValue** cmdlet, which returns the values for only named keys. For example, if you want to return the path to the Windows Defender executable defined by the **WindowsDefender** key, type the following command:

```
Get-ItemPropertyValue HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run -Name WindowsDefender
```

The **Registry** provider does not support the **Invoke-Item** cmdlet. If you attempt to run the **Invoke-Item** cmdlet by using a registry key location, it returns an error.

- Use the **Item** and **ChildItem** nouns to manage registry keys
- Use the **ItemProperty** noun to manage key values
- All verbs are supported except for **Invoke**
- ***-ItemProperty** commands support the **-Type** parameter to define the data type of the value
- Use **Get-ItemPropertyValue** to retrieve just key values

Studijní materiály Okskolení

The **Registry** provider creates a dynamic parameter, *-Type*, for the ***-ItemProperty** commands that are unique to the **Registry** provider. The following table lists valid parameter values and their equivalent registry data types.

Parameter value	Registry data type
String	REG_SZ
ExpandString	REG_EXPAND_SZ
Binary	REG_BINARY
DWord	REG_DWORD
MultiString	REG_MULTI_SZ
QWord	REG_QWORD
Unknown	Unsupported types such as REG_RESOURCE_LIST

The **Registry** provider supports transactions that allow you to manage multiple commands as a single unit. The commands in a transaction will either all be committed (completed) or the results will be rolled back (undone). This allows you to set multiple registry values at one time without the concern that some of the settings will update while others fail. Use the *-UseTransaction* parameter to include a command in a transaction.



Note: For more information about transactions in Windows PowerShell, refer to the [about_Transactions](#) help topic.



Note: It is important for you to remember to back up your registry settings before attempting to modify registry keys and values. To export registry settings to a file, use the **reg.exe** command.

Demonstration: Managing the registry

In this demonstration, you will see how to manage the registry by using the **Registry** provider in Windows PowerShell.

Demonstration Steps

1. Set the working folder location to the **Software** path within the **PSDrive** for **HKEY_Local_Machine**.
2. View the registry keys under **HKLM\Software**.
3. Create a registry key named **Demo** under the **HKLM\SOFTWARE** path.
4. Add the following value to the key that you just created:
 - Name: **Demo**
 - Value: **Test**
 - Type: **String**
5. View the registry value that you just created.

Working with certificates

If you have viewed or managed security certificates on a client or server computer, you have probably used the Certificates snap-in for **Microsoft Management Console**. The Certificates snap-in allows you to browse the certificates stores on the computer to which you connect.

The Windows PowerShell **Certificates** provider also allows you to view and manage security certificates. The **Certificates** provider creates a **PSDrive** named **Cert**.

The **Cert** drive always has at least two high-level store locations that group certificates for all users and the local computer, which are **CurrentUser** and **LocalMachine**.

The **Certificates** provider supports the **Get**, **Set**, **Move**, **New**, **Remove**, and **Invoke** verbs in combination with the **Item** and **ChildItem** nouns. The **ItemProperty** noun is not supported. In addition, all the ***-Location** commands are supported.

The **Invoke-Item** command will open the Certificates snap-in.

The **Get-ChildItem** command has a variety of dynamic parameters that are unique to the **Certificate** provider. These include:

- **-CodeSigningCert**. This gets certificates that can be used for code signing.
- **-DocumentEncryptionCert**. This gets certificates for document encryption.
- **-DnsName**. This gets certificates with the domain name in the **DNSNameList** property of the certificate. This parameter accepts wildcards.
- **-EKU**. This gets certificates with the specified text in the **EnhancedKeyUsageList** property. This parameter supports wildcards.
- **-ExpiringInDays**. This gets certificates that are expiring within the specified number of days.
- **-SSLServerAuthentication**. This gets only Secure Sockets Layer (SSL) web hosting certificates from a server.

- Use the **Item** or **ChildItem** noun
- Cannot use the **ItemProperty** noun
- **Get**, **Set**, **New**, **Move**, **Remove**, and **Invoke** verbs are supported
- **Invoke-Item** opens the Certificates snap-in
- **Get-ChildItem** has several unique parameters when used in the **Cert:** drive

Working with other PSDrives

In addition to the built-in providers for managing the file system, registry, and security certificates, Windows PowerShell provides other built-in providers. These are:

- **Alias**. Use this to view and manage Windows PowerShell aliases.
- **Environment**. Use this to view and manage Windows environment variables.
- **Function**. Use this to view and manage Windows PowerShell aliases.

- Built-in providers:
 - **Alias**
 - **Environment**
 - **Function**
 - **Variable**
 - **WSMan**
- Applications can have their own providers. Examples include:
 - **ActiveDirectory** (AD DS)
 - **WebAdministration** (IIS)

- **Variable.** Use this to view and manage Windows PowerShell variables.
- **WSMan.** Use this to view and manage WS-Management configurations.

In addition to the providers that are directly associated with Windows PowerShell, management tools for other applications can include their own providers. Examples include:

- **ActiveDirectory.** This provider is part of the **ActiveDirectory** module, which installs with Active Directory Domain Services (AD DS) and the Remote Server Administration Tools. The **ActiveDirectory** provider supports viewing and managing an AD DS database.
- **WebAdministration.** This provider is part of the **WebAdministration** module which installs with IIS, allowing you to view and manage application pools, websites, web applications, and virtual directories.

These additional providers support most, if not all, standard provider verbs and nouns. There might also be specific cmdlets that can perform the same functions. For instance, you can use the following provider-based command to return a list of all aliases in the current Windows PowerShell session:

```
Get-Item -Path Alias:
```

The previous command returns the same results as this dedicated command:

```
Get-Alias
```

However, there is no **Remove-Alias** command that deletes an alias. Therefore, if you want to delete an alias named **MyAlias**, you must use one of the following commands:

```
Remove-Item -Path Alias:MyAlias
```

or

```
Clear-Item -Path:MyAlias
```

Like the other providers that earlier topics covered, these **PSProviders** can have dynamic parameters or properties associated with them. The **Alias** provider, for example, includes the dynamic parameter **-Options**, which you can use to specify the **Options** property of an alias.

You should review the help for a provider before using it so that you can identify any dynamic parameters or properties. If you are not sure of the exact name of the provider and want to use wildcards to search for provider help, you can include the *-Category* parameter and the **Provider** value when you use the **Get-Help** command. To view the help for the **Alias** provider by using a wildcard, type the following command, and then press Enter:

```
Get-Help Alias* -Category Provider
```

Question: What are dynamic parameters and how do they relate to **PSProviders**?

Studijní materiály Okškolení

Lab: Using PSProviders and PSDrives

Scenario

You are a system administrator for the London branch office of Adatum Corporation. You must reconfigure several settings in your environment. You have recently learned about **PSProviders** and **PSDrives** and that you can access these settings by using a **PSProvider**. You have decided to use **PSDrives** to reconfigure these settings.

Objectives

After completing this lab, you will be able to:

- Create new items on a **PSDrive**.
- Create new **PSDrive** mappings.
- Create new item properties in a **PSDrive**.
- Modify items and properties in a **PSDrive**.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1** and **10961C-LON-CL1**

User name: **ADATUM\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available VM environment. Before you begin the lab, you must follow these steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the VM starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

The exercise steps should be performed on the **10961C-LON-CL1** VM throughout this lab.

Exercise 1: Creating files and folders on a remote computer

Scenario

You need to make sure that you can create folders and files on a remote computer so that you are able to copy automation scripts to the computer in the future. You have decided not to use the **MkDir** command or any of its aliases so that you are sure you are using the cmdlets that are associated with **PSProviders** and **PSDrives**.

The main tasks for this exercise are as follows:

1. Create a new folder on a remote computer.
2. Create a new PSDrive mapping to the remote file folder.
3. Create a file on the mapped drive.

► Task 1: Create a new folder on a remote computer

1. On **LON-CL1**, open **Windows PowerShell** as an administrator.
2. Read the complete help for the **New-Item** command.

Notice the **-Name** and **-ItemType** parameters, and then review the example commands.

3. Use **New-Item** to create a new folder (directory) named **C:\ScriptShare** on **\LON-SVR1\CS\$**.

► Task 2: Create a new PSDrive mapping to the remote file folder

1. In the **Windows PowerShell** console, read the complete help for the **New-PSDrive** command.
2. Create a new **PSDrive** named **ScriptShare**, and then map it to the **C:\ScriptShare** folder.

► Task 3: Create a file on the mapped drive

1. In the **Windows PowerShell** console, view the complete help for the **Set-Location** command.
2. Set the current working folder location to the mapped drive **ScriptShare**.
3. In the **ScriptShare** folder, create a text file named **script.txt**.
4. Confirm that the mapped drive contains the **script.txt** file.

Results: After completing this exercise, you should have successfully created a new folder and file on a remote computer and mapped a drive to that folder.

Exercise 2: Creating a registry key for your future scripts

Scenario

In this exercise, you will create a new registry key that you will use for storing configuration data for scripts that you will develop in the future. You will also create a registry setting in that key where you will store the name of the **PSDrive** you created previously. You want to verify that you can retrieve the value from the registry in scripts that you will create later.

The main tasks for this exercise are as follows:

1. Create the registry key to store script configurations.
2. Create a new registry setting to store the name of the PSDrive.

► **Task 1: Create the registry key to store script configurations**

1. In the **Windows PowerShell** console, type a command to verify that the registry key **HKEY_CURRENT_USER\Software** does not have a subkey named **Scripts**.
2. In the console, create a registry key named **Scripts** in **HKEY_CURRENT_USER\Software**.

► **Task 2: Create a new registry setting to store the name of the PSDrive**

1. In the **Windows PowerShell** console, type a command to set the current working location to the path of the registry key that you created.
2. Create the registry setting to store the **PSDrive** name with the following values:
 - Name: **PSDriveName**
 - Value: **ScriptShare**
3. Verify that you can retrieve the **PSDriveName** setting from the **HKey_Current_User\Software\Scripts** key.

Results: After completing this exercise, you should have successfully created a registry key to store configuration data and added a registry setting that stores the name of a **PSDrive**. You also should have verified that you can retrieve the value of that setting by using Windows PowerShell.

Exercise 3: Creating a new Active Directory group

Scenario

You need to create an Active Directory group named **London Developers**. In this exercise, you will create the group by using the **ActiveDirectory** provider and the **AD:** drive.

The main tasks for this exercise are as follows:

1. Create a PSDrive that maps to the Users container in AD DS.
2. Create the London Developers group.
3. Prepare for the next module.

► **Task 1: Create a PSDrive that maps to the Users container in AD DS**

1. In the **Windows PowerShell** console, load the **ActiveDirectory** module.
2. Create the following **PSDrive**:
 - Name: **AdatumUsers**
 - Root: **CN=Users,DC=Adatum,DC=com**
 - PSProvider: **ActiveDirectory**
3. Set the current working location to the new **PSDrive**.

► Task 2: Create the London Developers group

1. In the **Windows PowerShell** console, create the following group in AD DS:
 - o Name: **London Developers**
 - o Path: **CN=Users**
2. Verify that the new group was created.

Results: After completing this exercise, you should have successfully created an Active Directory group by using the **ActiveDirectory** provider.

► Task 3: Prepare for the next module

When you finish the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: Of the **PSProviders** that are included with Windows PowerShell, which support the use of alternative credentials?

Question: Windows PowerShell 3.0 and newer can make one kind of **PSDrive** visible in File Explorer. What kind of drive is that, and how do you make it visible?

Module Review and Takeaways

Review Question

Question: What is the advantage of managing a data store such as Active Directory by using a **PSProvider** instead of commands?

Real-world Issues and Scenarios

A parameter of the **Get-ChildItem** cmdlet does not work with a particular **PSDrive**. For example, **-Filter** does not work when listing information in a registry drive. This is a known issue, and it occurs because each **PSProvider** has different capabilities. The **Registry** provider does not support the **-Filter** parameter.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
A PSDrive that was present in Windows PowerShell is no longer present.	

Module 6

Querying management information by using CIM and WMI

Contents:

Module Overview	6-1
Lesson 1: Understanding CIM and WMI	6-2
Lesson 2: Querying data by using CIM and WMI	6-7
Lesson 3: Making changes by using CIM and WMI	6-15
Lab: Working with CIM and WMI	6-19
Module Review and Takeaways	6-23

Module Overview

Windows Management Instrumentation (WMI) and Common Information Model (CIM) both provide local and remote access to a repository of management information, including robust information available from the operating system, from computer hardware, and from installed software. In this module, you will learn about these two technologies.



Additional Reading: For more information on CIM cmdlets, refer to: "PowerShell Team Blog" at: <https://aka.ms/leh5h3>

Objectives

After completing this module, you will be able to:

- Explain the differences between CIM and WMI.
- Query management information by using CIM and WMI.
- Invoke methods by using CIM and WMI.

Lesson 1

Understanding CIM and WMI

In this lesson, you will learn about the architecture of both CIM and WMI. Both technologies connect to a common information repository that holds management information that you can query and manipulate. The repository contains all kinds of information about a computer system or device, including hardware, software, hardware drivers, components, roles, services, user settings, and just about every configurable item and the current state of that item. Knowing the framework and syntax of CIM and WMI will help you to know and control virtually every aspect of an operating system environment.

Lesson Objectives

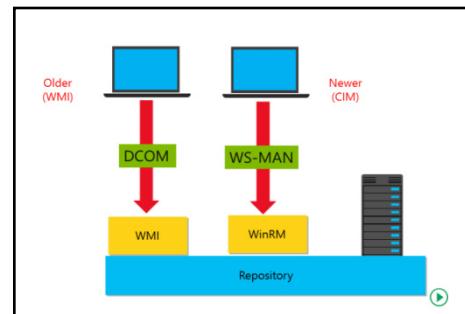
After completing this lesson, you will be able to:

- Describe the architecture of both CIM and WMI.
- Explain the purpose of the repository.
- Explain how to locate online documentation for repository classes.

Architecture and technologies

WMI and CIM are related technologies. Both are based on industry standards defined by the Distributed Management Task Force (DMTF), which defines independent management standards that can be implemented across different platforms or vendor enterprise environments. WMI, which is the Microsoft implementation of the Web-Based Enterprise Management standard, is an earlier technology that is based on preliminary standards and proprietary technology. CIM is a newer technology that is based on open, cross-platform standards. WMI has been available on the Windows operating system since Windows NT 4.0.

Both technologies provide a way to connect to a common information repository (also known as the *WMI repository*). This repository holds management information that you can query and manipulate. Windows PowerShell 3.0 and newer support both technologies. Earlier versions of Windows PowerShell support only WMI. In Windows PowerShell 3.0 and newer, two parallel sets of commands let you perform tasks by using either WMI or CIM.



Reference Links: You can access the DMTF website at: www.dmtf.org

CIM commands

CIM commands provide many cross-platform and cross-version capabilities. They support three kinds of connections:

- Connections to the local computer, which use the Distributed Component Object Model (DCOM) or the Web Services for Management (WS-MAN) protocol depending on the command you use.
- Ad-hoc connections to a remote computer, which always use the WS-MAN protocol. This protocol is based on HTTP.
- Session-based connections to a remote computer, which can use either DCOM or WS-MAN.

You typically make DCOM connections to the WMI service that is part of the Windows operating system. You make WS-MAN connections to the Windows Remote Management (WinRM) service, which is the same service that enables Windows PowerShell remoting. You will learn more about remoting in Module 10, "Administering remote computers." WinRM, which is part of the Windows Management Framework, is included in Windows Management Framework 2.0 and newer. WinRM is installed by default on computers running Windows 7 or newer or Windows Server 2008 R2 or newer. Although installed by default on all those operating systems, WinRM and remoting are enabled by default only on Windows 8 and newer and Windows Server 2012 and newer.

CIM commands don't require you to enable Windows PowerShell remoting and they don't require Windows PowerShell on the target computer. You can use CIM commands to work with Linux computers. You can also have CIM commands use the earlier WMI technology.



Note: Windows PowerShell on both Windows 10 and Windows Server 2016 comes with the Windows Management Framework 5.1. This version of Windows Management Framework has several improvements over version 5.0, and it includes the newest versions of Windows PowerShell, CIM, WMI, WinRM, and Software Inventory and Licensing components. You can download and install Windows Management Framework 5.1 for Windows 8.1, Windows 7, Windows Server 2012 R2, Windows Server 2012, and Windows Server 2008 R2.

WMI commands

WMI commands use the same repository as CIM commands. The only difference is how the WMI commands connect to a remote computer.

WMI commands do not support session-based connections. These commands support only ad-hoc connections over DCOM. Whether used by WMI or CIM commands, DCOM might be difficult to use on some networks. DCOM uses the remote procedure call (RPC) protocol. That protocol requires special firewall exceptions to work correctly.

WMI commands communicate with the WMI service. They do not require any version of the Windows Management Framework on a remote computer, and they do not require that Windows PowerShell remoting be enabled. If the remote computer has the **Windows Firewall** feature enabled, WMI commands require that the remote administration exceptions be enabled on the remote computer. If the remote computer has a different local firewall enabled, equivalent exceptions must be created and enabled.

Should you use CIM or WMI?

You should use CIM cmdlets instead of the older WMI cmdlets, because:

- CIM cmdlets use DCOM when querying the local computer.
- CIM cmdlets use WS-MAN for remote access.
- CIM cmdlets can use DCOM or WS-MAN for session-based connections to remote computers.

Studijní materiály Oskolení

- CIM uses CIM sessions for accessing multiple devices.
- CIM improves the way of dealing with WMI associations.
- You can use the **Get-CimClass** cmdlet for investigating WMI classes.

The CIM cmdlets can connect to computers that do not have Windows Management Framework 2.0 or newer installed and to computers that do not have Windows PowerShell remoting enabled. You must use a **CimSession** object to connect to those computers. You will learn about CIM sessions in the next lesson.

 **Note:** Microsoft considers the WMI commands within Windows PowerShell to be deprecated, although the underlying WMI repository is still a current technology. You should rely primarily on CIM commands, and use WMI commands only when CIM commands are not practical.

Using commands instead of classes

The repository is not well documented, so discovering the classes that you need to perform a specific task might be difficult and impractical. The solution is to use Windows PowerShell commands that behave like any other shell command but that internally use CIM or WMI. This approach gives you the advantages of Windows PowerShell commands, such as discoverability and built-in documentation, while also giving you the existing functionality of the repository.

In Windows 8 and Windows Server 2012, Microsoft introduced hundreds of new commands in Windows PowerShell. Windows 10 and Windows Server 2016 include many more. Many of these commands internally use CIM or WMI. These commands provide better access to the functionality of CIM and WMI so that you can use that functionality without having to deal with their complexity.

Understanding the repository

The repository that CIM and WMI use is organized into namespaces. A *namespace* is a folder that groups related items for organizational purposes.

Namespaces contain classes. A *class* represents a manageable software or hardware component. For example, the Windows operating system provides classes for processors, disk drives, services, user accounts, and so on. Each computer on the network might have slightly different namespaces and classes. For example, a domain controller might have a class named **ActiveDirectory** that does not exist on other computers.

- The repository used by CIM and WMI is organized into namespaces
- Namespaces organize related classes:
 - Toggle through the namespaces by using tab completion in the **Get-CimInstance -Namespace** cmdlet
- Classes represent manageable components
- An instances is an actual occurrence of a class
- An instances has
 - Properties that describe the instance's attributes
 - Methods that cause the instance perform an action

To find the top-level namespaces, type the following in the **Windows PowerShell** console:

```
Get-CimInstance -Namespace root -ClassName Namespace
```

You get the following list of namespaces:

- **root/AccessLogging**
- **root/Appv**
- **root/aspnet**

- **root/CIMV2**
- **root/Cli**
- **root/DEFAULT**
- **root/directory**
- **root/Hardware**
- **root/HyperVCluster**
- **root/Interop**
- **root/InventoryLogging**
- **root/Microsoft**
- **root/msdtc**
- **root/PEH**
- **root/Policy**
- **root/RSOP**
- **root/SECURITY**
- **root/SecurityCenter2**
- **root/ServiceModel**
- **root/StandardCimv2**
- **root/subscription**
- **root/virtualization**
- **root/WMI**



Note: This list is not complete. You get additional namespaces depending on the role and services installed on a Windows Server 2016 computer.

When you work with the repository, you typically work with instances. An *instance* is an actual occurrence of a class. For example, if your computer has two processor sockets, you will have two instances of the class that represents processors. If your computer does not have an attached tape drive, you will have zero instances of the tape drive class.

Instances are objects, similar to the objects that you have already used in Windows PowerShell. Instances have properties, and some instances have methods. *Properties* describe the attributes of an instance. For example, a network adapter instance might have properties that describe its speed, power state, and so on. *Methods* tell an instance to do something. For example, the instance that represents the operating system might have a method to restart the operating system.

Finding documentation

Finding class documentation might be difficult, but you can find a lot of information on the Microsoft Developer Network (MSDN) website at:

<https://aka.ms/vgw7v0> and in the TechNet

Microsoft Script Center at: <https://aka.ms/xu24li>.

Although many Microsoft product groups and independent software vendors expose management information in the repository, only a few of them create formal documentation. In most cases, an Internet search for a class name provides your best option for finding the documentation that exists.

The classes in the **root\CMIV2** namespace comprise an exception. The MSDN Library typically documents these classes well. However, an Internet search provides the best way to locate the documentation for a particular class. For example, typing the class name **Win32_OperatingSystem** into an Internet search engine is the fastest way to locate the documentation webpage for that class.

Remember that CIM and WMI are not native parts of Windows PowerShell. Instead, they are external technologies that Windows PowerShell can use and understand. However, because they are external technologies, Windows PowerShell Help does not document the repository classes.

- The fastest way to find documentation is to type a repository class name into an Internet search engine
- Classes in the **root\CMIV2** namespace are typically well documented
- Classes from other namespaces are typically not well documented

Demonstration: Finding documentation for classes

In this demonstration, you will see how to locate online class documentation.

Demonstration Steps

- Using the host computer, locate the online documentation for the **Win32_BIOS** class.

Question: Can you think of any situations for which you have to use WMI instead of CIM?

Lesson 2

Querying data by using CIM and WMI

In this lesson, you will use both CIM and WMI commands to query the repository. As Microsoft transitions from using only WMI commands to primarily using CIM commands, you will benefit from learning the differences.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to write a command that lists the available namespaces.
- Explain how to list local repository namespaces.
- Explain how to retrieve a list of classes from a namespace.
- Explain how to retrieve a list of classes from the **root\CIMv2** namespace and to sort them.
- Explain how to query instances of a specified class.
- Explain how to query instances of a specified class by using WMI, CIM, and WMI Query Language (WQL).
- Explain how to connect to remote computers by using CIM or WMI.
- Explain how to create and manage CIM sessions.
- Explain how to query repository classes from remote computers by using **CimSession** objects.

Listing namespaces

You can use Windows PowerShell to list all the namespaces on either the local or a remote computer. To list the namespaces, run the following command:

- Listing namespaces helps you discover what the repository on your computer contains
- To list the namespaces, run the following command:
 - `Get-WmiObject -Namespace root -List -Recurse | Select -Unique __NAMESPACE`
 - CIM commands offer tab completion for the `-Namespace` parameter

```
Get-WmiObject -Namespace root -List -Recurse | Select -Unique __NAMESPACE
```

 **Note:** It can take several minutes to produce a complete list of namespaces. Notice that `__NAMESPACE` has two underscores.

Windows PowerShell also supports tab completion for namespace names when you use CIM commands. In this module, you will primarily use the **root\CLIMv2** namespace, which includes all the classes related to the Windows operating system and your computer's hardware. The **root\CLIMv2** namespace is the default namespace. Therefore, you do not have to specify the namespace when querying instances from it unless otherwise noted.

Demonstration: Listing local repository namespaces by using WMI

In this demonstration, you will see how to list local repository namespaces by using WMI.

Demonstration Steps

- Use WMI to list all the local namespaces.

Listing classes

Windows PowerShell can list all the classes in a particular namespace. For example, to list all the classes in the **root\CLIMv2** namespace, run either of these commands:

```
Get-WmiObject -Namespace root\CLIMv2 -List
Get-CimClass -Namespace root\CLIMv2
```



Note: It can take several minutes to list all the classes in a large namespace such as **root\CLIMv2**.

Windows PowerShell supports the tab completion of namespace names only for CIM commands. You can type **Get-Cim[tab] -Nam[tab] roo[tab]** to quickly type the second command by pressing the Tab key where **[tab]** is shown.

Windows PowerShell does not list classes in any particular order. You can more easily find a class when they are listed alphabetically. For example, if you want a class that represents a process but do not know the class name, you can quickly move to the "P" section of a sorted list and start to look for the word *process*. To produce an alphabetical list of classes in the **root\CLIMv2** namespace, run either of these commands:

```
Get-WmiObject -Namespace root\cimv2 -List | Sort Name
Get-CimClass -Namespace root\CLIMv2 | Sort CimClassName
```

- Listing classes in alphabetical order can make it easier to decide whether the class you need exists
- To produce an alphabetical list of classes in the **root\CLIMv2** namespace, run either of following commands:
 - `Get-WmiObject -Namespace root\cimv2 -List | Sort Name`
 - `Get-CimClass -Namespace root\CLIMv2 | Sort CimClassName`



Note: In the **root\CMv2** namespace, you will see some class names that start with **Win32**_ and others that start with **CIM_**. This namespace is the only one that uses these prefixes. Classes that start with **CIM_** are typically abstract classes. Classes that start with **Win32_** are typically more-specific versions of the abstract classes, and they contain information specific to the Windows operating system.

Many administrators feel that the repository is difficult to work with. Finding the class that you need to perform a particular task is basically a guessing game. You have to guess what the class might be named and then look through the class list to determine whether you are right. Then you must query the class to determine whether it contains the information that you need. Because many classes outside the **root\CMv2** namespace are not well documented, this is your best approach. An administrator who is good at using WMI and CIM is also good at making educated guesses.

No central directory of repository classes exists. The repository does not include a search system. You can use Windows PowerShell to perform a basic keyword search of repository class names. For example, to find all the classes in the **root\CMv2** namespace having *network* in the class name, use the following command:

```
Get-CimClass *network* | Sort CimClassName
```

However, this technique cannot search class descriptions, because that information is not stored in the repository. An Internet search engine provides a better way to search for possible class names.



Note: You might see some classes names that begin with two underscores (_). These are system classes that WMI and CIM use internally.



Reference Links: A graphical WMI Explorer tool, written in a Windows PowerShell script, is available at: <https://aka.ms/cu41zp>. This tool can make it easier to explore the WMI classes that are available on a particular computer.

There is one specific WMI class object that can cause problems for system administrators. This is the **Win32_Product** class. This class is not query optimized. When you query this class, the provider performs a Windows Installer (MSI) reconfiguration on every MSI package on the system as the query is being performed. You can see the effect of this by opening the application event log. You will see several Windows Installer messages showing each installed application being reconfigured. A reconfiguration performs a revalidation on the application's installation, which usually includes an MSI repair if there is an inconsistency between the package and the original MSI file.



Note: The Microsoft Support page for this MSI reconfiguration issue is at:
<https://aka.ms/jlgark>

Demonstration: Listing and sorting the classes from a namespace

In this demonstration, you will see how to list and sort the classes in a namespace.

Demonstration Steps

1. List the classes in the **root\SecurityCenter2** namespace.
2. List the classes in the **root\CIMv2** namespace. Sort the list by name.
3. List all the classes in the **root\CIMv2** namespace that have *network* in the class name.

Querying instances

When you know the class you want to query, Windows PowerShell can retrieve class instances for you. For example, if you want to retrieve all instances of the **Win32_LogicalDisk** class from the **root\CIMv2** namespace, run either of the following commands:

```
Get-WmiObject -Class Win32_LogicalDisk
Get-CimInstance -ClassName Win32_LogicalDisk
```



Note: The default configuration settings in Windows PowerShell cause these two commands to have differently formatted output.

Both the **-Class** parameter of **Get-WmiObject**, and the **-ClassName** parameter of **Get-CimInstance** are positional. That means the following commands work the same way:

```
Get-WmiObject Win32_LogicalDisk
Get-CimInstance Win32_LogicalDisk
```

By default, both commands retrieve all available instances of the specified class. You can specify filter criteria to retrieve a smaller set of instances. The filter languages used by these commands do not use Windows PowerShell comparison operators. Instead, they use traditional programming operators, as shown in the following table.

Comparison	WMI and CIM operator	Windows PowerShell operator
Equality	=	-eq
Inequality	<>	-ne
Greater than	>	-gt
Less than	<	-lt

Comparison	WMI and CIM operator	Windows PowerShell operator
Wildcard string match	LIKE (with % as the wildcard)	-like (with * as the wildcard)
Less than or equal to	<=	-le
Greater than or equal to	>=	-ge
Boolean AND	AND	-and
Boolean OR	OR	-or

For example, to retrieve only the instances of **Win32_LogicalDisk** for which the **DriveType** property is 3, run either of the following commands:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3"
Get-CimInstance -ClassName Win32_LogicalDisk -Filter "DriveType=3"
```

 **Note:** Many class properties use integers to represent different kinds of things. For example, in the **Win32_LogicalDisk** class, a **DriveType** property of 3 represents a local fixed disk. A value of 5 represents an optical disk, such as a DVD drive. You have to examine the class documentation to learn what each value represents.

Querying by using WQL

Both WMI and CIM accept query statements written in WQL. These query statements are commonly used in Microsoft Visual Basic Scripting Edition (VBScript) scripts. Here is an example:

```
Get-WmiObject -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType = 3"
Get-CimInstance -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType = 3"
```

This command syntax makes it easy to reuse existing query statements that you have or that you find in examples written by other people. A detailed examination of WQL is beyond the scope of this course.

Demonstration: Querying class instances

In this demonstration, you will see several ways to query class instances from the repository.

Demonstration Steps

1. Use WMI to display all instances of the **Win32_Service** class.
2. Use CIM to display all instances of the **Win32_Process** class.
3. Use CIM to display those instances of the **Win32_LogicalDisk** class that have a drive type of 3.
4. Use CIM and a WQL query to display all instances of the **Win32_NetworkAdapter** class.

Studijní materiály Oskolení

Connecting to remote computers

You can use both the WMI and the CIM commands to connect to a remote computer. When you connect to a remote computer, you can also specify alternative credentials for the connection.

 **Note:** You do not have to specify alternative credentials for connections to the local computer.

For the WMI commands, use the *-ComputerName* parameter to specify a remote computer's name or IP address. You can specify multiple computer names, either in a comma-separated list or by providing a parenthetical command that produces a collection of computer names as string objects. Use the *-Credential* parameter to specify an alternative user name. In this case, you will be prompted for the password. Here is an example:

```
Get-WmiObject -ComputerName LON-DC1 -Credential ADATUM\Administrator -Class Win32_BIOS
```

If you specify multiple computer names, Windows PowerShell will contact them one at a time in the order that you specify. If one computer fails, the command will produce an error message and continue to try the remaining computers.

The CIM commands also support a *-ComputerName* parameter. You can use these to create an ad-hoc connection. If you plan to query multiple classes from the same computer, you can achieve better performance by creating a persistent CIM session instead.

You can run the following CIM command to achieve the same results as the **Get-WmiObject** command in the preceding code example, but this command does not accept the *-Credential* parameter:

```
Get-CimInstance -ClassName Win32_BIOS -ComputerName LON-DC1
```

The preceding command retrieves the CIM instances of a class named **Win32_BIOS** from **LON-DC1**, accepting the credentials of the signed-in user, **Adatum\Administrator**.

Remember that the CIM commands use the WS-MAN protocol for ad-hoc connections. This protocol has specific authentication requirements. Between computers in the same domain or in trusting domains, you typically have to provide a computer's name as it appears in Active Directory Domain Services. You cannot provide an alias name or an IP address. You will learn more about these and other restrictions in Module 10, "Administering remote computers." You will also learn how to work around these restrictions.

- You use *-ComputerName* to query from a remote computer
- You use *-Credential* to specify an alternate credential for remote connections using WmiObject only
`Get-WmiObject -ComputerName LON-DC1 -Credential ADATUM\Administrator -Class Win32_BIOS`
- The CIM equivalent does not use *-Credential*
`Get-CimInstance -ClassName Win32_BIOS -ComputerName LON-DC1`
- WMI uses DCOM
- CIM uses WinRM for ad-hoc connections

Using CIM sessions

A *CIM session* is a persistent connection to a remote computer, made by using the WS-MAN or DCOM protocol. After a session is created, you can use it to process multiple queries for that computer. You will achieve better performance across multiple queries by using a session rather than by using multiple ad-hoc connections.

The basic syntax to create a session and store it in a variable is:

- CIM sessions:
 - Are reusable, persistent, and authenticated connections to a remote computer
 - Can be created and stored in a variable
 - Allow you to pass a CIM session object in the *-CimSession* parameter instead of using *-ComputerName* to target the computer in the specified session
 - Can be manually closed when no longer needed

```
$s = New-CimSession -ComputerName LON-DC1
```

You can create multiple sessions at the same time:

```
$s2 = New-CimSession -ComputerName LON-CL1,LON-DC1
```

When you have one or more sessions in a variable, you can send CIM queries to those sessions:

```
Get-CimInstance -CimSession $s2 -ClassName Win32_OperatingSystem
```

Remember that sessions are designed to work best in a domain environment, between computers in the same domain or in trusting domains. If you have to create a session to a nondomain computer or to a computer in an untrusted domain, you will need to do additional configuration. You will learn more about that configuration in Module 10, “Administering remote computers.”

A session option allows you to specify many session settings. The following option, for example, lets you create the session by using DCOM instead of WS-MAN:

```
$opt = New-CimSessionOption -Protocol Dcom
$sess = New-CimSession -ComputerName LON-DC1 -SessionOption $opt
Get-CimInstance -ClassName Win32_BIOS -CimSession $sess
```

The first line in the preceding code defines the CIM DCOM session option. The second line defines the session variable by using that CIM DCOM session option, and the final line returns data from the remote computer by using that defined session.

CIM sessions remain open while being used. You can also manually close the sessions for a specified remote computer:

```
Get-CimSession -ComputerName LON-DC1 | Remove-CimSession
```

To close a session that you have stored in a variable, use the following command:

```
$sess | Remove-CimSession
```

To close all open sessions, use the following command:

```
Get-CimSession | Remove-CimSession
```



Note: The Help for some commands, such as **Get-SmbShare**, state that they support a *-CimSession* parameter. Those commands use CIM internally. When you use those commands to query a remote computer, you can provide a CIM session object to the *-CimSession* parameter to connect by using an existing session.

Demonstration: Using CIMSession objects

In this demonstration, you will see how to query repository classes from remote computers by using **CimSession** objects.

Demonstration Steps

1. Create a CIM session to **LON-DC1**.
2. Store the session in a variable.
3. By using the variable, query the **Win32_OperatingSystem** class.
4. Close the session.

Question: What are the advantages of creating and using CIM sessions instead of ad-hoc connections?

Lesson 3

Making changes by using CIM and WMI

In this lesson, you will learn to use CIM and WMI to make changes by running methods. Several methods are available to you, depending on whether you use WMI or CIM. Knowing these methods is an important step in querying and manipulating the repository information.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to discover the methods of repository objects.
- Explain how to find online documentation for methods.
- Explain how to find the methods of the **Win32_Service** class and how to find online documentation for the **Change** method of **Win32_Service**.
- Explain how to invoke methods of repository objects.
- Explain how to invoke the **Reboot** method of **Win32_OperatingSystem** and the **Terminate** method of **Win32_Process**.

Discovering methods

You have learned that objects contain members and that members have properties, methods, and events associated with them.

A method tells an object to perform an action or task. Repository objects that you query by using CIM or WMI have methods that reconfigure the manageable components that the objects represent. For example, the **Win32_Service** class instances represent background services. The class has a **Change** method that reconfigures many of a service's settings, including the sign-in password, name, and start mode.

If you know the class representing the manageable component that you want to reconfigure, you can discover the methods of that class by using **Get-Member**:

```
Get-WmiObject -Class Win32_Service | Get-Member
```

The resulting list displays all the available methods. Remember that not every class offers methods, so the list might not include any methods. The output of **Get-Member** does not explain how to use the methods, so unless you already know how to use them, you have to find the documentation for the class.

You can find the static methods of a Microsoft .NET Framework class by using **Get-Member**. **Get-Member** has a parameter named **-Static**, which directs the command to return the static methods and properties of a type. For example, the following command returns all the methods and properties for the **DateTime** .NET Framework class:

```
[DateTime] |Get-Member -Static
```

- Many repository classes include methods
- A method tells an object to perform a task or action
- Repository class methods typically reconfigure the manageable component that the class represents
- Use **Get-Member** to discover the methods of a class
- Note that the output does not explain how to use a method, so you need documentation for that

The same technique does not work with **Get-CimInstance**, because the objects have only a set of methods but no properties. Instead, you must run the following command:

```
Get-CimClass -Class Win32_Service | Select-Object -ExpandProperty CimClassMethods
```

To do this in Windows PowerShell, you have to query every class, pipe each one to **Get-Member**, and then search the output for a method name or keyword. This approach is very time consuming and impractical. An Internet search engine provides a faster and easier way to search for classes and methods.

Finding documentation for methods

If any method documentation exists, the documentation webpage for the repository class will include it. Remember that repository classes are not typically well documented, especially the classes that are not in the **root\CIMv2** namespace.

An Internet search engine provides the fastest way to find the documentation for a class. Type the class name into a search engine, and one of the first few search results will typically point to the class documentation page. Generally, the documentation page includes a section named “Methods,” which lists the class methods. Click any method name to display the instructions for using that method.

Methods	
The Win32_Service class has these methods.	
Method	Description
Change	Modifies a service.
ChangeStartMode	Modifies the start mode of a service.
Create	Creates a new service.
Delete	Deletes an existing service.
GetSecurityDescriptor	Returns the security descriptor that controls access to the service.
InterrogateService	Requests that a service update its state to the service manager.
PauseService	Attempts to place a service in the paused state.
ResumeService	Attempts to place a service in the resumed state.
SetSecurityDescriptor	Writes an updated version of the security descriptor that controls access to the service.
StartService	Attempts to place a service into the startup state.
StopService	Places a service in the stopped state.
UserControlService	Attempts to send a user-defined control code to a service.

The documentation mainly allows you to determine the arguments that each method requires. For example, the **Win32Shutdown** method of the **Win32_OperatingSystem** class accepts a single argument. This argument is an integer, and it tells the method to either shut down, restart, or sign out.

Demonstration: Finding methods and documentation

In this demonstration, you will see how to find class methods and how to find their documentation.

Demonstration Steps

1. Display the members of the **Win32_Service** class.
2. Use Windows PowerShell to display the definition of the **Change** method of **Win32_Service**.
3. Run the equivalent command by using **Get-CimClass**. Sort the list by the **Name** property.
4. Run the **Get-CimClass** command by using the parameters from the first command in step 1.
5. On the host machine, open Internet Explorer and find online documentation for the **Change** method of **Win32_Service**.

Invoking methods

When you know the method you want to use and how to use it, you can invoke the method. Three ways exist to invoke a method. The following sections describe these ways.

Using the **Invoke-WmiMethod** command

You can use the **Invoke-WmiMethod** command by itself, or you can use the pipeline to send it a WMI object from **Get-WmiObject**. Here are two examples that work the same way:

- The three ways to invoke a method are:
 - **Invoke-WmiMethod**
 - **Invoke-CimMethod**
 - **ForEach-Object**
- Both **Invoke** techniques can be used with or can accept a pipeline object from the corresponding **Get** command
- The returned object includes a *ReturnValue* parameter:
 - Zero typically means success
 - For other values, see the documentation

```
Get-WmiObject -Class Win32_OperatingSystem |
Invoke-WmiMethod -Name Win32Shutdown -Argument 0
Invoke-WmiMethod -Class Win32_OperatingSystem -Name Win32Shutdown -Argument 0
```

Both **Get-WmiObject** and **Invoke-Method** have the *-ComputerName* parameter that lets you run the method on a remote computer.

Invoke-WmiMethod will not work correctly if the argument list has to contain a null value. For example, the **Change** method of the **Win32_Service** class accepts several arguments. You can provide null as the value for any argument that you do not want to change. For example, if you want to change only the service's sign-in password, you provide null for the first seven arguments and a new password for the eighth argument. **Invoke-WmiMethod** does not support the use of those null values.

Invoke-WmiMethod is a WMI command. That means that it communicates by using the DCOM protocol.



Note: WMI implements a system of privileges in addition to the usual permissions. A **privilege** helps to protect sensitive operations, such as restarting a computer. To run those sensitive operations, your WMI connection must be enabled for privileges. The **-EnableAllPrivileges** switch of **Get-WmiObject** enables privileges, and you must use it when you plan to invoke a sensitive method.

Using the **Invoke-CimMethod** command

The **Invoke-CimMethod** command resembles **Invoke-WmiMethod**. However, because it is a CIM command, it communicates by using different protocols:

- When you connect to the local repository, it uses DCOM.
- When you connect to a remote computer, it uses WS-MAN.
- When you use an established CIM session, it uses either DCOM or WS-MAN depending on how the session was created.

The argument list for **Invoke-CimMethod** is a dictionary object. Such an object consists of one or more key-value pairs. The key for each pair is the argument name, and the value for each pair is the corresponding argument value. Here is an example:

```
Invoke-CimMethod -ComputerName LON-DC1 -Class Win32_Process -MethodName Create -Arguments
@{ 'Path' = 'Notepad.exe' }
```

To include multiple arguments in the dictionary, use a semicolon (;) to separate each key-value pair. The command can also accept a repository object from **Get-CimInstance**:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name='notepad.exe'" | Invoke-CimMethod -MethodName Terminate
```

Notice that you do not have to specify the *-Arguments* parameter for methods that do not require any arguments.

If you use *-ComputerName* or *-CIMSession* with **Get-CimInstance** and pipe the resulting object to **Invoke-CimMethod**, **Invoke-CimMethod** will invoke the method on whatever computer or session the object came from. For example, to terminate a process on a remote computer, you can run the following command:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name='notepad.exe'" -Computername LON-DC1 | Invoke-CimMethod -MethodName Terminate
```

Using **ForEach-Object**

If you cannot use **Invoke-WmiMethod** or **Invoke-CimMethod**, you can retrieve repository objects and enumerate them to run methods. Here is an example:

```
Get-WmiObject -Class Win32_Service -Filter "Name='MyService'" |  
ForEach-Object { $PSItem.Change($null,$null,$null,$null,$null,$null,"P@ssw0rd") }
```

The preceding example changes the sign-in password of the service named **MyService**. When you use this technique to invoke a method, you must follow these rules:

- You must follow the method name by an open parenthesis. Do not include a space between the method name and the parenthesis.
- You must include the parentheses even if the method does not accept any arguments.
- You provide the arguments in a comma-separated list.
- Windows PowerShell uses the built-in variable *\$null* to represent the value null.

The equivalent **Get-CimInstance** command will not work for the preceding example, because **[Microsoft.Management.Infrastructure.CimInstance]** does not contain a method named **Change**.

Demonstration: Invoking methods of repository objects

In this demonstration, you will see how to invoke methods of repository objects.

Demonstration Steps

1. Using CIM and the **Reboot** method of **Win32_OperatingSystem**, restart **LON-DC1**.
2. Start **Microsoft Paint**.
3. Using WMI and the **Terminate** method of **Win32_Process**, close Paint.

Question: What are some disadvantages of using **ForEach-Object** instead of one of the **Invoke** commands to invoke a method?

Lab: Working with CIM and WMI

Scenario

You have to query management information from several computers. You start by querying the information from your local computer and from one test computer in your environment.

Objectives

After completing this lab, you will be able to:

- Query information by using WMI commands.
- Query information by using CIM commands.
- Invoke methods by using WMI and CIM commands.

Lab Setup

Estimated Time: **45 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **ADATUM\Administrator**

Password: **Pa55w.rd**

The changes that you make during this lab will be lost if you revert your virtual machines at another time during the class.

For this lab, you will use the available virtual machine environment. Before you begin the lab, complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Perform the lab steps on the **10961C-LON-CL1** virtual machine.

Exercise 1: Querying information by using WMI

Scenario

In this exercise, you will discover repository classes and then use WMI commands to query them.

The main tasks for this exercise are as follows:

1. Query IP addresses.
2. Query operating-system version information.
3. Query computer-system hardware information.
4. Query service information.

► **Task 1: Query IP addresses**

1. On **LON-CL1**, start **Windows PowerShell** as an administrator.
2. Note that an IP address is part of a network adapter's configuration. Using the keyword **configuration**, find a repository class that lists IP addresses being used by the local computer.
3. Using a WMI command and the class that you discovered in the previous step, display a list of all the IP addresses received by using Dynamic Host Configuration Protocol (DHCP).

► **Task 2: Query operating-system version information**

1. Using the keyword **operating**, find a repository class that lists operating-system version information. Sort it by the **name** property.
2. Display a list of properties for the class that you discovered in the previous step.
3. Make note of the properties that contain the operating-system version, the service pack major version, and the operating-system build number.
4. Using a WMI command and the class that you discovered in step 1, display the local operating-system version, service pack major version, and operating-system build number.

► **Task 3: Query computer-system hardware information**

1. Using the keyword **system**, find a repository class that contains computer-system information.
2. Display a list of properties and property values for the class that you discovered in the previous step.
3. Using the list of properties and a WMI command, display the local computer's manufacturer, model, and total physical memory. Label the column for total physical memory **RAM**.

► **Task 4: Query service information**

1. Using the keyword **service**, find a repository class that contains service information.
2. Display a list of properties and property values for the class that you discovered in the previous step.
3. Using the list of properties and a WMI command, display the service name, status (Running or Stopped), and sign-in name for all services that have names starting with *S*.

Results: After completing this exercise, you should have queried repository classes by using WMI commands.

Exercise 2: Querying information by using CIM

Scenario

In this exercise, you will discover new repository classes and query them by using CIM commands.

The main tasks for this exercise are as follows:

1. Query user accounts.
2. Query BIOS information.
3. Query network adapter configuration information.
4. Query user group information.

► **Task 1: Query user accounts**

1. Using a WMI command and the keyword **user**, find a repository class that lists user accounts.
2. Using a CIM command, display a list of properties for the class that you discovered in the previous step.
3. Using a CIM command and the property list, display a list of user accounts in a table. Include columns for the account caption, domain, security ID, full name, and name. The full name column might be blank for some or all accounts.

► **Task 2: Query BIOS information**

1. Using the keyword **bios** and a WMI command, find a repository class that contains BIOS information.
2. Using a CIM command and the class that you discovered in the previous step, display a list of all available BIOS information.

► **Task 3: Query network adapter configuration information**

1. Use a CIM command to display all the local instances of the **Win32_NetworkAdapterConfiguration** class.
2. Use a CIM command to display all instances of the **Win32_NetworkAdapterConfiguration** class that exist on **LON-DC1**.

► **Task 4: Query user group information**

1. Using a WMI command and the keyword **group**, find a class that lists user groups.
2. Using a CIM command, display a list of the user groups that exist on **LON-DC1**.

Results: After completing this exercise, you should have queried repository classes by using CIM commands.

Exercise 3: Invoking methods

Scenario

In this exercise, you will use WMI and CIM commands to invoke methods of repository objects.

The main tasks for this exercise are as follows:

1. Invoke a CIM method.
2. Invoke a WMI method.
3. Prepare for the next module.

► **Task 1: Invoke a CIM method**

- Using a CIM command and the **Reboot** method of **Win32_OperatingSystem**, restart **LON-DC1**.

► Task 2: Invoke a WMI method

1. Switch back to the **LON-CL1** virtual machine.
2. Using WMI commands and the **ChangeStartMode** method of **Win32_Service**, change the start mode of the WinRM service to Automatic.

Results: After completing this exercise, you should have used CIM and WMI commands to invoke methods of repository objects.

► Task 3: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Question: One of your lab tasks directed you to query **Win32_Product**. Do you know of any disadvantages of using this class?

Question: What are the main differences between WMI and CIM?

Module Review and Takeaways

Review Question

Question: What do you think is the most difficult part about working with WMI and CIM?

Real-world Issues and Scenarios

Most organizations have deployed Windows Management Framework 3.0 or newer. This means that you should use the CIM commands for ad-hoc connections in all environments. Microsoft is encouraging the use of CIM over WMI whenever possible.

Tools

Tool	Description	Where to find it
PowerShell Scriptomatic	A graphical tool for exploring the repository	https://aka.ms/fhb9y0
A WMI Explorer tool	A tool that provides the ability to browse and view WMI namespaces, classes, instances, and properties in a single pane of view	https://aka.ms/kfxmrj

Best Practice

Use CIM commands whenever possible. Compared to WMI commands, CIM commands offer better performance, and Microsoft continues to develop and improve the CIM commands over time.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You get an RPC server not found error when you use WMI commands.	
You get errors when you use CIM to connect to a remote computer by using the WS-MAN protocol.	
You get an access denied error when attempting to connect to a remote computer.	

Studijní materiály Okškolení

Module 7

Working with variables, arrays, and hash tables

Contents:

Module Overview	7-1
Lesson 1: Using variables	7-2
Lesson 2: Manipulating variables	7-9
Lesson 3: Manipulating arrays and hash tables	7-15
Lab: Working with variables	7-23
Module Review and Takeaways	7-27

Module Overview

Variables are an essential component of scripts. You can use variables to accomplish complex tasks that you cannot by using a single command. In this module, you will learn how to work with variables, arrays, and hash tables as one of the steps in learning how to write Windows PowerShell scripts.

Objectives

After completing this module, you will be able to:

- Assign a value to variables.
- Describe how to manipulate variables.
- Describe how to manipulate arrays and hash tables.

Lesson 1

Using variables

In this lesson, you will learn about variables and some rules on how to use them. This is important to ensure that the data you store in variables is in the correct format and easily accessible. When working with variables, it is important to name them appropriately and to assign them the right data type.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of variables.
- Describe the naming rules for using variables.
- Explain how to assign a value to a variable.
- Describe variable types.
- Explain how to assign a variable type.

What are variables?

When using the Windows PowerShell pipeline, you can pass data through the pipeline and perform operations on it. This lets you perform many bulk operations such as querying a list of objects, filtering the objects, and modifying the objects or displaying the data. The main limitation of the pipeline is that the process flows only in one direction and it is difficult to perform complex operations. You can use variables to solve this problem. *Variables* store values and objects in memory so that you can perform complex and repetitive operations on them.

Some of the things you can do with a variable are:

- Store the name of a log file that you write data to multiple times.
- Derive and store an email address based on the name of a user account.
- Calculate and store the date of a day 30 days prior to the current day, to identify whether computer accounts have signed in during the last 30 days.

In addition to simple data types such as numbers or strings, variables can hold objects too. When a variable contains an object, you can access all of the object's characteristics. For example, if you store an Active Directory user object in a variable, all of the properties of that user account are stored in the variable as well, and you can view them.

You can view the variables contained in memory by viewing the contents of the PSDrive named Variable:

`Get-ChildItem Variable:`

You can also view the variables in memory by using the **Get-Variable** cmdlet:

`Get-Variable`

- A variable stores a value or object in memory
- Some things you can do with a variable:
 - Store the name of a log file that you write data to multiple times
 - Derive and store an email address based on the name of a user account
 - Calculate and store the date of the day 30 days prior to the current day to identify whether computer accounts have signed in during the last 30 days
 - You can access object properties when stored in a variable
 - Variables and their values can be viewed in the PSDrive named Variable:



Note: Windows PowerShell includes multiple cmdlets for creating, manipulating, and viewing variables, but these are seldom used because you can create and manipulate variables without using them. Therefore, the cmdlets for manipulating variables are mentioned only briefly in this course.

Variable naming

Like many other aspects of computing, a good variable name is one that is easy to understand. You should create variable names that describe the data stored in them. For example, a variable that stores a user account could be `$user`, and a variable that stores the name of a log file could be `$logFile`.

In most cases, you will see variables used with a dollar sign (\$) symbol. The \$ symbol is not part of the variable name, but is an indicator to Windows PowerShell that it is a variable. For example, `$user` is a variable named user, and the \$ symbol helps Windows PowerShell to identify that it is a variable.

You should typically limit variable names to alphanumeric characters (letters and numbers). While you can include some special characters and spaces, it becomes more confusing to use. To include a space in a variable name, you need to enclose the name in braces ({}). For example, `${log File}` includes a space.

Variable names are not case sensitive. The variable `$USER` and `$user` are interchangeable. For readability, the common convention is to use lowercase characters and capitalize the first letter of the words in a variable name. Capitalization of the first word is optional depending on the situation and your preferences. For example, `$logFile` and `$LogFile` are both commonly used. However, when variables are used for parameters in a script, the first word should be capitalized for consistency with the parameters used by cmdlets. Using a capital letter acts as a separator between the words and makes the variable name readable without using special characters such as spaces, hyphens, or underscores.

- Variable names:
 - Should be easily understandable
 - Can contain spaces if enclosed in braces
 - Should contain only alphanumeric characters
 - Are not case sensitive
- A common convention for variable names uses capital letters to separate words:
 - `$LogFile`
 - `$StartDate`
 - `$IpAddress`

Assigning a value to a variable

You use the standard mathematical operators when working with variable values. These are the operators equals (=), plus (+), and (minus (-)) that you are already familiar with.

To assign a value to a variable, you use the = operator. For example:

- Use standard mathematical operators when working with variables
- To assign a value to a variable, use =
 - `$num1 = 5`
 - `$LogFile = "C:\Logs\Log.txt"`
 - `$user = Get-ADUser Administrator`
 - `$service = Get-Service W32Time`
- To display the value of a variable:
 - `$num1`
 - `Write-Host "The log location is $logfile"`
- To clear a variable, use `$null`:
 - `$num1 = $null`

```
$num1 = 10
$LogFile = "C:\Logs\Log.txt"
```

Studyjní materiály pro školení

You can also put values into a variable by using a command that is evaluated. The result of the command is placed in the variable. For example:

```
$user = Get-ADUser Administrator  
$service = Get-Service W32Time
```

 **Note:** If a variable is assigned values from a command that returns multiple values or objects, then the variable becomes an array that contains multiple values. You will learn about arrays later in this module.

You can display the value of a variable by typing the variable name and then pressing Enter. You can also display the value as part of a command by using **Write-Host**. For example:

```
$user  
Write-Host "The location of the log file is $LogFile"
```

 **Note:** When you display a variable by using **Write-Host**, the variable is evaluated and the value is displayed only when you use double quotes. If you use single quotes, the variable is not evaluated and only the name of the variable is displayed.

To remove all values from a variable, you can set the variable equal to `$null`. The `$null` variable is automatically defined by Windows PowerShell as being nothing. For example:

```
$num1 = $null  
$str1 = $null
```

 **Note:** To clear a variable, you can also use **Clear-Variable**.

You can use mathematical operators with variables. For example:

```
$area = $length * $width  
$sum = $num1 + $num2  
$path = $folder + $file
```

You can set the value of a variable by using the **Set-Variable** cmdlet. When you use this cmdlet, you do not include the `$` symbol when referring to the name. For example:

```
Set-Variable -Name num1 -Value 5
```

Variable types

All variables are assigned a type. The type of a variable determines the data that can be stored in it. In most cases, you allow Windows PowerShell to automatically assign the type to the variable when you assign the value. Automatic assignment of the variable type works well most of the time. However, in some cases, the data type is ambiguous, and you might prefer to specify the variable type.

The following table lists some common variable types used in Windows PowerShell.

- Type of a variable determines the data that can be stored in it
- String: Stores text, including special characters
- Int32: Stores integers without decimals
- Double: Stores numbers with decimals
- DateTime: Stores date and time
- Bool: Stores true or false
- Windows PowerShell can automatically assign the type based on a value
- Specify the type if data is going to be ambiguous

Type	Description
String	A string variable stores text that can include special characters. For example, "This is a string."
Int	A 32-bit integer variable stores a number without decimal places. For example, 228.
Double	A 64-bit floating point variable stores a number that can include decimal places. For example, 128.45.
DateTime	A DateTime variable stores a date object that includes a date and time. For example, January 5, 2018 10:00 AM.
Bool	A Boolean variable can store only the values \$true or \$false.

If you do not assign a variable type, Windows PowerShell assigns a type automatically based on what you put in the variable. When the value is contained in quotes, it is generally interpreted as a string. For example, Windows PowerShell would interpret 5 as an integer but would interpret "5" as a string.

You can force a variable to accept only a specific type of content by defining the type. When you define the type, Windows PowerShell attempts to convert the value you provide into the correct type. If Windows PowerShell is unable to convert the value into the correct type, it returns an error.

The examples below show the \$num2 variable being defined as a 32-bit integer and the \$date variable being defined as DateTime:

```
[Int]$num2 = "5"
[DateTime]$date = "January 5, 2018 10:00AM"
```

You can view a variable's type by appending the **GetType()** method to the name of the variable. For example:

```
$date.GetType()
```

Demonstration: Assigning a variable type

In this demonstration, you will learn how to assign values and types to variables.

Demonstration Steps

1. On **LON-CL1**, click the **Start** button, and then type **powersh**.
2. In the results list, right-click **Windows PowerShell** and then click **Run as administrator**.
3. To set the value of `$num1` to 5, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$num1 = 5
```

4. To display the value of `$num1`, type the following command, and then press Enter:

```
$num1
```

5. To set the value of `$logFile` to be C:\Logs\Log.txt, type the following command, and then press Enter:

```
$logFile = "C:\Logs\Log.txt"
```

6. To view the value of `$logFile`, type the following command, and then press Enter:

```
$logFile
```

7. To set the value of `$service` to the W32Time service, type the following command, and then press Enter:

```
$service = Get-Service W32Time
```

8. To display the value of `$service`, type the following command, and then press Enter:

```
$service
```

9. To display `$logFile` as part of a text message on screen, type the following command, and then press Enter:

```
Write-Host "The log file location is $logFile"
```

10. To view all of the properties of the service object stored in `$service`, type the following command, and then press Enter:

```
$service | Format-List *
```

11. To view the **Status** property of `$service`, type the following command, and then press Enter:

```
$service.Status
```

12. To view the **Name** and **Status** properties of `$service`, type the following command, and then press Enter:

```
$service | Format-Table Name,Status
```

13. To view the variables in memory, type the following command, and then press Enter:

```
Get-Variable
```

14. To view the variables in memory, type the following command, and then press Enter:

```
Get-ChildItem Variable:
```

15. To view the variable type of \$num1, type the following command, and then press Enter:

```
$num1.GetType()
```

16. To view the variable type of \$logFile, type the following command, and then press Enter:

```
$logFile.GetType()
```

17. To view the variable type of \$service, type the following command, and then press Enter:

```
$service.GetType()
```

18. To view the properties and methods of \$service, type the following command, and then press Enter:

```
$service | Get-Member
```

19. To set the value of \$num2 as a string of 5, type the following command, and then press Enter:

```
$num2 = "5"
```

20. To set the value of \$num3 as a 32-bit integer of 5, type the following command, and then press Enter:

```
[Int]$num3 = "5"
```

21. To verify the variable type of \$num2, type the following command, and then press Enter:

```
$num2.GetType()
```

22. To verify the variable type of \$num3, type the following command, and then press Enter:

```
$num3.GetType()
```

23. To set \$date1 as a string, type the following command, and then press Enter:

```
$date1 = "March 5, 2019 11:45 PM"
```

24. To set \$date2 as a DateTime type, type the following command, and then press Enter:

```
[DateTime]$Date2 = "March 5, 2019 11:45 PM"
```

25. To verify the variable type of \$date1, type the following command, and then press Enter:

```
$date1.GetType()
```

26. To verify the variable type of \$date2, type the following command, and then press Enter:

```
$date2.GetType()
```

27. To attempt to convert a string to a 32-bit integer, type the following command, and then press Enter:

```
[Int]$num4 = "Text that can't convert"
```

28. To view how variable types can convert during operations, type the following command, and then press Enter:

```
$num2 + $num3
```

29. To view how variable types can convert during operations, type the following command, and then press Enter:

```
$num3 + $num2
```

30. To view how variable types can fail to convert during operations, type the following command, and then press Enter:

```
$num3 + $logFile
```

31. Close the Windows PowerShell prompt.

Question: Why is it sometimes necessary to assign a variable type?

Lesson 2

Manipulating variables

In this lesson, you will learn how to manipulate the contents of variables. Each type of variable has a unique set of properties that you can access. Each variable also has a unique set of methods that you can use to manipulate it. You can identify the properties and methods for use in your scripts.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to identify methods and properties.
- Explain how to work with strings.
- Explain how to manipulate strings.
- Explain how to work with dates.
- Explain how to manipulate dates.

Identifying methods and properties

Just as objects in Windows PowerShell have properties and methods, so do variables. The properties and methods that you can use vary depending on what is stored in the variable. If a variable contains an object such as an Active Directory Domain Services (AD DS) user account or a Windows service, then the properties and methods for the variable match those for the object. If a variable is the DateTime type, then the properties and methods for DateTime are available.

The simplest method for identifying the properties and methods available for a variable is to pipe the variable to the **Get-Member** cmdlet. The **Get-Member** cmdlet displays all the available properties and methods for a specific variable based on its variable type. For example:

```
$LogFile | Get-Member
```

- Properties and methods for a variable are based on the type of variable
- To identify the properties and methods for a variable, use:
 - Get-Member
 - Tab completion
- Documentation for properties and methods is available in the .NET Framework Class Library

To browse through the properties and methods for a variable, you can use tab completion by typing the name of the variable appended with a dot. When you press Tab, the properties and methods available for the variable display.

When you view the properties and methods for a variable, some of them will be easily understandable. If you do not understand how to use a property or method for a variable, then you can review documentation for that variable type in the Microsoft .NET Framework Class Library. Each variable type has its own section in the documentation. For example, the documentation for Decimal variables is located in Decimal Structure.



Additional Reading: For more information on .NET Framework variable types, refer to: "System Namespace" at: <https://aka.ms/krlgav>

Studijní materiály Okškolení

The documentation for the .NET Framework is oriented towards developers and sometimes it can be difficult to understand. You can also search the Internet for examples relating to a specific method or property. Many examples are available online.

Working with strings

String variables are commonly used in scripts. They can be used to store user input and other text data. There are many methods you can use to manipulate strings. Many of the methods are seldom used, but it is good to be aware of them in case you ever need them.

The string variable has only one property, **Length**. When you view the length for a string variable, it returns the number of characters in the string. For example:

```
$logFile.Length
```

The following table lists some of the methods available for string variables.

Method	Description
Contains(string value)	Identifies whether the variable contains a specific string. The result is either \$true or \$false.
Insert(int startindex,string value)	Inserts a string of text at the character number specified.
Remove(int startindex,int count)	Removes a specified number of characters from the string beginning at the character number specified. If the count is not specified, then the string is truncated at specified character number.
Replace(string value,string value)	Replaces all instances of the first string with the second string.
Split(char separator)	Splits a single string into multiple strings at points specified by a character.
ToLower()	Converts a string to lowercase.
ToUpper()	Converts a string to uppercase.

- The only property available for strings is **Length**

- Some commonly used methods for strings are:
 - Contains(string value)
 - Insert(int startindex,string value)
 - Remove(int startindex,int count)
 - Replace(string value,string value)
 - Split(char separator)
 - ToLower()
 - ToUpper()

Demonstration: Manipulating strings

In this demonstration, you will learn how to manipulate string variables.

Demonstration Steps

1. On **LON-CL1**, right-click the **Start** button, and then click **Windows PowerShell (Admin)**.
2. To set `$logFile` with a value, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$logFile = "C:\Logs\log.txt"
```

3. To identify whether `$logFile` contains the text **C:**, type the following command, and then press Enter:

```
$logFile.Contains("C:")
```

4. To identify whether `$logFile` contains the text **D:**, type the following command, and then press Enter:

```
$logFile.Contains("D:")
```

5. To insert the text **\MyScript** at character 7, type the following command, and then press Enter:

```
$logFile.Insert(7, "\MyScript")
```

6. To verify that the value stored in `$logFile` has not changed, type the following command, and then press Enter:

```
$logFile
```

7. To update the value of `$logFile`, type the following command, and then press Enter:

```
$logFile=$logFile.Insert(7, "\MyScript")
```

8. To verify that the value of `$logFile` has changed, type the following command, and then press Enter:

```
$logFile
```

9. To replace **.txt** with **.htm**, type the following command, and then press Enter:

```
$logFile.Replace(".txt", ".htm")
```

10. To split the value of `$logFile` at the **** character, type the following command, and then press Enter:

```
$logFile.Split("\")
```

11. To view only the last item from the split, type the following command, and then press Enter:

```
$logFile.Split("\") | Select -Last 1
```

12. To convert the value to uppercase letters, type the following command, and then press Enter:

```
$logFile.ToUpper()
```

13. To convert the value to lowercase letters, type the following command, and then press Enter:

```
$logFile.ToLower()
```

14. Close the Windows PowerShell prompt.

Working with dates

Many scripts that you create will need to reference the current date or a previous point in time. For example, you might want to create a log file name based on the current date to ensure uniqueness. Additionally, you might be looking for objects in AD DS that have not signed in for an extended period of time. You can use DateTime variables to accomplish these tasks.

DateTime properties

A DateTime variable contains both the date and the time. You can use the properties of a DateTime variable to access specific parts of the date or time.

The following table lists some of the properties available for a DateTime variable.

Property	Description
Hour	Returns the hours of the time in 24-hour format.
Minute	Returns the minutes of the time.
Second	Returns the seconds of the time.
TimeOfDay	Returns detailed information about the time of day, including hours, minutes, and seconds.
Date	Returns only the date and not the time.
DayOfWeek	Returns the day of the week, such as Monday.
Month	Returns the month as a number.
Year	Returns the year.

DateTime methods

A DateTime variable also has many methods available that allow you to manipulate the time. Methods provide ways to add or subtract time. There also are methods to manipulate the output of a DateTime variable in specific ways. The table below lists some of the methods available for a DateTime variable.

Method	Description
AddDays(double value)	Adds the specified number of days.
AddHours(double value)	Adds the specified number of hours.
AddMinutes(double value)	Adds the specified number of minutes.
AddMonths(int months)	Adds the specified number of months.
AddYears(int value)	Adds the specified number of years.
ToLongDateString()	Returns the date in long format as a string.

Method	Description
ToShortDateString()	Returns the date in short format as a string.
ToLongTimeString()	Returns the time in long format as a string.
ToShortTimeString()	Returns the time in short format as a string.

 **Note:** If you need to subtract time from a DateTime variable, use one of the methods for adding time with a negative number. For example, \$date.AddDays(-60).

Demonstration: Manipulating dates

In this demonstration, you will learn how to manipulate DateTime variables.

Demonstration Steps

1. On **LON-CL1**, right-click the **Start** button, and then click **Windows PowerShell (Admin)**.
2. To put the current date and time in the variable `$date`, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$date = Get-Date
```

3. To display the value of `$date`, type the following command, and then press Enter:

```
$date
```

4. To display the **Hour** property of `$date`, type the following command, and then press Enter:

```
$date.Hour
```

5. To display the **Minute** property of `$date`, type the following command, and then press Enter:

```
$date.Minute
```

6. To display the **Day** property of `$date`, type the following command, and then press Enter:

```
$date.Day
```

7. To display the **DayOfWeek** property of `$date`, type the following command, and then press Enter:

```
$date.DayOfWeek
```

8. To display the **Month** property of `$date`, type the following command, and then press Enter:

```
$date.Month
```

9. To display the **Year** property of `$date`, type the following command, and then press Enter:

```
$date.Year
```

10. To add 100 days to `$date`, type the following command, and then press Enter:

```
$date.AddDays(100)
```

11. To subtract 60 days from \$date, type the following command, and then press Enter:

```
$date.AddDays(-60)
```

12. To display \$date as a long date string, type the following command, and then press Enter:

```
$date.ToString("yyyy-MM-dd HH:mm:ss")
```

13. To display \$date as a short date string, type the following command, and then press Enter:

```
$date.ToString("MM/dd/yyyy")
```

14. To display \$date as a long time string, type the following command, and then press Enter:

```
$date.ToString("HH:mm:ss")
```

15. To display \$date as a short time string, type the following command, and then press Enter:

```
$date.ToString("mm:ss")
```

16. Close the Windows PowerShell prompt.

Question: Why is it important to understand how to use **Get-Member** when manipulating variables?

Lesson 3

Manipulating arrays and hash tables

Arrays and hash tables allow you to store more complex data than just simple variables. Using arrays and hash tables allows you to complete more complex tasks with your scripts. In this lesson, you will learn how to use arrays and hash tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of an array.
- Describe how to work with arrays and their contents.
- Describe how to work with arraylists and their contents.
- Explain how to manipulate arrays and arraylists.
- Explain the purpose of a hash table.
- Describe how to work with hash tables and their contents.
- Explain how to manipulate hash tables.

What is an array?

An array is a data structure that is designed to store a collection of items. You can think of an array as a variable that contains multiple values or objects. Variables that contain a single value are useful, but for complex tasks you often need to work with groups of items. For example, you might need to update the Voice over IP (VoIP) attribute on multiple user accounts. Or, you might need to check the status of a group of services and start any that are stopped. When you put multiple objects or values into a variable, it becomes an array.

You can create an array by providing multiple values in a comma-separated list. For example:

```
$computers = "LON-DC1", "LON-SRV1", "LON-SRV2"
$numbers = 228, 43, 102
```

 **Note:** To create an array of strings, you put quotes around each item. If you put one set of quotes around all the items, it is treated as a single string.

You also can create an array by using the output from a command. For example:

```
$users = Get-ADUser -Filter *
$files = Get-ChildItem C:\
```

You can verify whether a variable is an array by using the **GetType()** method on the variable. The **BaseType** listed will be **System.Array**.

- An array contains multiple values or objects
- Values can be assigned by:
 - Providing a list


```
$computers = "LON-DC1", "LON-SRV1", "LON-CL1"
```
 - Using command output


```
$users = Get-ADUser -Filter *
```
- You can create an empty array:
 - \$newUsers = @()
- You can force an array to be created when adding only a single item:
 - [Array]\$computers = "LON-DC1"

Studijní materiály Okškolení

You can create an empty array before you are ready to put content in it. This can be useful when you have a loop later on in a script that adds items to the array. For example:

```
$newUsers = @()
```

You also can force an array to be created when adding a single value to a variable. This creates an array with a single value into which you can add items later. For example:

```
[array]$computers="LON-DC1"
```

Working with arrays

In your scripts, you will need to refer to the data that you place in arrays. You can either access all items in the array at once, or access them individually. To display all items in an array, you type the variable name and then press Enter, just as you would for a variable with a single value.

You can refer to individual items in an array by their index number. When you create an array, each item is assigned an index number starting at zero. So, the first item placed in the array is item zero and the second item in the array is item 1. To display a specific item, place the index number in brackets after the array name. The following example displays the first item in an array:

```
$users[0]
```

You also can add a new item to an array. The following example adds the user account stored in \$user1 to the \$users array:

```
$users = $users + $user1
```

Often you will see this syntax shortened as the following example shows:

```
$users += $user1
```

To identify what you can do with the content in an array, use the **Get-Member** cmdlet. Pipe the contents of the array to **Get-Member**, and the results returned identify the properties and methods that you can use for the items in the array. For example:

```
$files | Get-Member
```



Note: When you pipe an array containing mixed data types to **Get-Member**, results are returned for each data type. Therefore, this is also a way of determining which data types are in the array.

To view the properties and methods available for an array rather than the items in the array, use this syntax:

```
Get-Member -InputObject $files
```

- To display all items in an array:
 - \$users
- To display specific items in an array by using an index number:
 - \$users[0]
- To add items to an array:
 - \$users = \$users + \$user1
 - \$users += \$user1
- To pipe the array to **Get-Member** to identify what you can do with the array contents
 - \$files | Get-Member

Working with arraylists

The default type of array that Windows PowerShell creates is a fixed-size array. This means that when you add an item to the array, the array is actually recreated with the additional item. When you work with relatively small arrays, this is not a concern. However, if you add thousands of items to an array one by one, its performance reduces. The other concern when using fixed-size arrays is removing items. There is no simple method to remove an item from a fixed-size array.

To address the shortcomings of arrays, you can use an arraylist. An arraylist functions similar to an array except that it does not have a fixed size. This means that you can use methods to add and remove items.

To create an arraylist when assigning values, use the following syntax:

```
[System.Collections.ArrayList]$computers = "LON-DC1", "LON-SRV1", "LON-CL1"
```

To create an arraylist that is empty and ready to add items, use the following syntax:

```
$computers=New-Object System.Collections.ArrayList
```

When you use an arraylist, you can use methods to both add and remove items. However, these methods will fail when you try to use them on a fixed-size array. For example:

```
$computers.Add("LON-SRV2")
$computers.Remove("LON-CL1")
```

 **Note:** When you remove an item from an arraylist, if there are multiple matching items, then only the first instance is removed.

If you want to remove an item from an arraylist based on the index number, you use the **RemoveAt()** method. For example:

```
$computers.RemoveAt(1)
```

Demonstration: Manipulating arrays and arraylists

In this demonstration, you learn how to manipulate arrays and arraylists.

Demonstration Steps

1. On **LON-CL1**, right-click the **Start** button, and then click **Windows PowerShell (Admin)**.
2. To set `$computers` to be an array of strings, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computers = "LON-DC1", "LON-SRV1", "LON-CL1"
```

- Arrays are fixed size, which limits performance and makes removing items difficult
- Arraylists are variable sized
 - To create an arraylist:
 - `$computers = New-Object System.Collections.ArrayList`
 - `[System.Collections.ArrayList]$computers = "LON-DC1","LON-SRV1"`
 - To add or remove items from an arraylist:
 - `$computers.Add("LON-SRV2")`
 - `$computers.Remove("LON-CL1")`
 - `$computers.RemoveAt(1)`

3. To set `$users` to be an array of user objects, type the following command, and then press Enter:

```
$users = Get-ADUser -Filter *
```

4. To view the contents of the `$computers` array, type the following command, and then press Enter:

```
$computers
```

5. To view the contents of the `$users` array, type the following command, and then press Enter:

```
$users
```

6. To view the number of items in `$users`, type the following command, and then press Enter:

```
$users.Count
```

7. To view the user object at index 125 of `$users`, type the following command, and then press Enter:

```
$users[125]
```

8. To view the properties and methods available for the items in `$computers`, type the following command, and then press Enter:

```
$computers | Get-Member
```

9. To view the properties and methods available for the items in `$users`, type the following command, and then press Enter:

```
$users | Get-Member
```

10. To view the **UserPrincipalName** property for a user object in the array, type the following command, and then press Enter:

```
$users[125].UserPrincipalName
```

11. To add an item to `$computers`, type the following command, and then press Enter:

```
$computers += "LON-SRV2"
```

12. To verify that the item was added, type the following command, and then press Enter:

```
$computers
```

13. To create an arraylist containing user objects, type the following command, and then press Enter:

```
[System.Collections.ArrayList]$usersList = Get-ADUser -Filter *
```

14. To identify whether `$usersList` has a fixed size, type the following command, and then press Enter:

```
$usersListFixedSize
```

15. To view the number of items in `$arrayList`, type the following command, and then press Enter:

```
$usersList.Count
```

16. To view a single item in `$arrayList`, type the following command, and then press Enter:

```
$usersList[125]
```

17. To remove an item in \$arrayList, type the following command, and then press Enter:

```
$usersList.RemoveAt(125)
```

18. To verify that the item count is reduced by one, type the following command, and then press Enter:

```
$usersList.Count
```

19. To verify that the item at index 125 has changed, type the following command, and then press Enter:

```
$usersList[125]
```

20. Close the Windows PowerShell prompt.

What is a hash table?

A *hash table* is a similar concept to an array in that it stores multiple items. However, unlike an array, which uses an index number to identify each item, a hash table uses a unique key for each item. The key is a string that is a unique identifier for the item. Each key in a hash table is associated with a value.

The following table shows how an array can store a list of IP addresses.

- A hash table is a list of names and values
 - To refer to a value in the hash table, you provide the key:
- \$servers.'LON-DC1'
 - \$servers['LON-DC1']

Key	IP address
LON-DC1	172.16.0.10
LON-SRV1	172.16.0.11
LON-SRV2	172.16.0.12

Index number	Value
0	172.16.0.10
1	172.16.0.11
2	172.16.0.40

If the array is named \$ip, then you access the first item in the array by using:

```
$ip[0]
```

You can use a hash table to store both IP addresses and the computer names as shown in the following table.

Key	Value
LON-DC1	192.168.0.10
LON-SRV1	192.168.0.11
LON-SRV2	192.168.0.12

If the hash table is named `$servers`, then you access the first item in the hash table by using any of the following options:

```
$servers.'LON-DC1'  
$servers['LON-DC1']
```

 **Note:** You only need to place single quotation marks around the key for a hash table item if it contains a special character. In the example above, the hyphen in the computer names is a special character and this requires the key name to be enclosed in single quotation marks.

Working with hash tables

Working with hash tables is similar to working with an array, except that to add items to a hash table, you need to provide both the key for the item and the value. The following command creates a hash table named `$servers` to store server names and IP addresses:

- To define a hash table:
 - `$servers = @{"LON-DC1" = "172.16.0.10"; "LON-SRV1" = "172.16.0.11"}`
- To add an item to a hash table:
 - `$servers.Add("LON-SRV2","172.16.0.12")`
- To remove an item from a hash table:
 - `$servers.Remove("LON-DC1")`
- To update a value for an item in a hash table:
 - `$servers.'LON-SRV2' = "172.16.0.100"`

```
$servers = @{"LON-DC1" = "172.16.0.10"; "LON-SRV1" = "172.16.0.11"}
```

Notice the following syntax in the example above:

- It begins with the `@` symbol.
- The keys and associated values are enclosed in braces.
- The items are separated by a semicolon.

 **Note:** The semicolon between hash table items is required because they are all on the same line. If you place each item on a separate line, the semicolons are not required as separators.

Adding or removing items from a hash table is similar to an arraylist. You use the methods **Add()** and **Remove()**. For example:

```
$servers.Add("LON-SRV2", "172.16.0.12")  
$servers.Remove("LON-DC1")
```

You can also update the value for a key. For example:

```
$servers.'LON-SRV2' = "172.16.0.100"
```

To view all properties and methods available for a hash table, use the **Get-Member** cmdlet. For example:

```
$servers | Get-Member
```

Demonstration: Manipulating hash tables

In this demonstration, you learn how to manipulate hash tables.

Demonstration Steps

1. On **LON-CL1**, right-click the **Start** button, and then click **Windows PowerShell (Admin)**.
2. To create a hash table with the names of users and a department for each, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$users = @{"Lara"="IT"; "Peter"="Managers"; "Sang"="Sales"}
```

3. To view the contents of the hash table, type the following command, and then press Enter:

```
$users
```

4. To view the department for a single user, type the following command, and then press Enter:

```
$users.Lara
```

5. To update the department for a user, type the following command, and then press Enter:

```
$users.Sang = "Marketing"
```

6. To verify that the department was updated, type the following command, and then press Enter:

```
$users
```

7. To add a new user, type the following command, and then press Enter:

```
$users.Add("Tia", "Research")
```

8. To remove a user, type the following command, and then press Enter:

```
$users.Remove("Sang")
```

9. To verify the added and removed users, type the following command, and then press Enter:

```
$users
```

10. To create a new hash table for a calculated property, type the following command, and then press Enter:

```
$prop = @{n="Size(KB)"; e={$_['Length/1KB]}}
```

11. To view the hash table, type the following command, and then press Enter:

```
$prop
```

12. To view the name and size of the files in **C:\Windows**, type the following command, and then press Enter:

```
Get-ChildItem C:\Windows -File | Format-Table Name,Length
```

13. To view the size of files by using the calculated property, type the following command, and then press Enter:

```
Get-ChildItem C:\Windows -File | Format-Table Name,$prop
```

14. Close the Windows PowerShell prompt.

Question: Why is it important to understand the difference between an array and an arraylist?

Lab: Working with variables

Scenario

You are preparing to begin writing scripts to automate server administration in your organization. Before you begin creating scripts, you want to practice working with variables, arrays, and hash tables.

Objectives

After completing this lab, you will be able to:

- Work with variable types.
- Use arrays.
- Use hash tables.

Lab Setup

Estimated Time: **45 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before beginning the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Microsoft Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Exercise 1: Working with variable types

Scenario

You first plan to practice working with different types of variables.

The main tasks for this exercise are as follows:

1. Use string variables.
2. Use DateTime variables.

► Task 1: Use string variables

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. Create a variable `$logPath` that contains **C:\logs**.
3. For `$logPath`, identify the type of variable and the available properties and methods.
4. Create a variable `$logFile` that contains **log.txt**.

5. Update `$logPath` to include the contents of `$logFile`.
6. Update the path stored in `$logPath` to use drive **D** instead of drive **C**.
7. Leave the Windows PowerShell prompt open for the next task.

► Task 2: Use DateTime variables

1. At the Windows PowerShell prompt, create a variable `$today` that contains today's date.
2. For `$today`, identify the type of variable and the available properties and methods.
3. Use the properties of `$today` to create a string in the format **Year-Month-Day-Hour-Minute.txt**, and store the value in `$logFile`.
4. Create a variable `$cutOffDay` that is **30 days** before today.
5. Use **Get-ADUser** to query user accounts that have signed in since `$cutOffDay`. Filter by using the **LastLogonDate** property.
6. Leave the Windows PowerShell prompt open for the next exercise.

Results: After completing this exercise, you should have manipulated multiple types of variables.

Exercise 2: Using arrays

Scenario

Now that you practiced using different types of variables, you want to work with arrays.

The main tasks for this exercise are as follows:

1. Use an array to update the department for users.
2. Use an arraylist.

► Task 1: Use an array to update the department for users

1. Query all Active Directory Domain Services (AD DS) users in the **Marketing** department and place them in a variable named `$mktgUsers`. Include the **Department** property in the results.
2. Use `$mktgUsers` to identify the number of users in the Marketing department.
3. Display the first user in `$mktgUsers` and verify that the **Department** property is listed.
4. Pipe the users in `$mktgUsers` to **Set-ADUser** and update the department to **Business Development**.
5. View the **Department** attribute in `$mktgUsers` to see if it has been updated.
6. Query all AD DS users in the Marketing department to verify that there are none.
7. Query all AD DS users in the Business Development department and verify that it matches the previous count from the Marketing department.
8. Leave the Windows PowerShell prompt open for the next task.

► Task 2: Use an arraylist

1. Create an arraylist named `$computers` with the values **LON-SRV1**, **LON-SRV2**, and **LON-DC1**.
2. Verify that `$computers` does not have a fixed size.
3. Add **LON-DC2** to `$computers`.

4. Remove **LON-SRV2** from `$computers`.
5. Display the contents of `$computers`.
6. Leave the Windows PowerShell prompt open for the next exercise.

Results: After completing this exercise, you should have manipulated arrays and arraylists.

Exercise 3: Using hash tables

Scenario

After using variables and arrays, you plan to practice working with hash tables.

The main tasks for this exercise are as follows:

1. Use a hash table.
2. Prepare for the next module.

► Task 1: Use a hash table

1. Create a hash table named `$mailList` with the following users and email addresses:
 - o Frank - Frank@fabrikam.com
 - o Libby - LHayward@contoso.com
 - o Matej - MStojanov@tailspintoys.com
2. Display the contents of `$mailList`.
3. Display the email address for **Libby**.
4. Update the email address for **Libby** to **Libby.Hayward@contoso.com**.
5. Add a new email address for **Stela**: **Stela.Sahiti@treyresearch.net**.
6. View the count of items in `$mailList`.
7. Remove **Frank** from `$mailList`.
8. Verify that **Frank** is removed.
9. Close the Windows PowerShell prompt.

Results: After completing this exercise, you should have manipulated a hash table.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: In the “Using arrays” exercise, why did user objects in `$mktgUsers` not update with the new department name?

Question: In Exercise 1, you replaced **C:** with **D:** in the variable `$logPath`. Why is it better to include colon than simply replace **C** with **D**?

Module Review and Takeaways

Review Questions

Question: You have queried a list of computers from AD DS and placed them into a variable. When you attempt to remove one of the computers from the variable by using the **Remove()** method, there is an error. What is the most likely cause of this error?

Question: You have placed the value "February 20, 2018" into a variable. What type of variable will it be?

Studijní materiály Okškolení

Studijní materiály Okškolení

Module 8

Basic scripting

Contents:

Module Overview	8-1
Lesson 1: Introduction to scripting	8-2
Lesson 2: Scripting constructs	8-11
Lesson 3: Importing data from files	8-17
Lab: Basic scripting	8-22
Module Review and Takeaways	8-28

Module Overview

Windows PowerShell commands allow you to do some powerful tasks such as modify thousands of user accounts by using a single command. Additionally, Windows PowerShell allows you to create scripts that you can use to accomplish more complex tasks that are not possible by using a single command. You can reuse scripts multiple times to accomplish repetitive tasks. Once the necessary commands are in a script, there is no risk of typing errors when you run it. In this module, you will learn to run, modify, and create scripts. You will also learn how to import data from a file.

Objectives

After completing this module, you will be able to:

- Run a Windows PowerShell script.
- Use Windows PowerShell scripting constructs.
- Import data from a file.

Lesson 1

Introduction to scripting

Most administrators start working with scripts by either downloading or modifying scripts that are created by others. After you have obtained a script, you must run it. There are some important differences between running scripts in Windows PowerShell versus running them at a standard command prompt. These differences are covered in this lesson along with other concepts that are related to scripting.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe Windows PowerShell scripts.
- Explain how to find scripts and modify them.
- Describe how to create scripts.
- Describe the PowerShellGet module.
- Explain how to run Windows PowerShell scripts.
- Describe the script execution policy.
- Explain how to set script execution policy.
- Explain how AppLocker can be used to secure Windows PowerShell scripts.
- Explain how to digitally sign scripts.
- Explain how to digitally sign a Windows PowerShell script.

Overview of Windows PowerShell scripts

You may first start using Windows PowerShell to accomplish tasks that cannot be done with graphical tools. For example, when managing Microsoft Office 365 or Microsoft Exchange Server, there are many settings that can only be configured by using Windows PowerShell cmdlets. As you become more familiar with Windows PowerShell, you will see opportunities to simplify management by using scripts instead of running individual commands.

You can use scripts to standardize repetitive tasks. Standardizing a task reduces the risk of errors. A script that has been tested can be run multiple times without errors, but when you manually type a command multiple times, there is a risk of error each time. Also, if the task must be performed on a scheduled basis, you can schedule the script to run as required.

- Windows PowerShell scripts can be used for:
 - Repetitive tasks
 - Complex tasks
 - Reporting
- Windows PowerShell scripts have a .ps1 file extension



Note: Configuring Windows PowerShell scripts to run as scheduled tasks or scheduled jobs is covered in Module 11, "Using background jobs and scheduled jobs."

You can also use scripts to accomplish more complex tasks than are practical by using a single command. It is technically possible to make a single Windows PowerShell command that is long and complex but is impractical to manage. Placing complex tasks in a script makes editing simpler and easier to understand.

Reporting is one complex and repetitive task that you can do with Windows PowerShell. You can use Windows PowerShell to create text or HTML-based reports. For example, you can create a script that reports available disk space on your servers, or you can create a script for Exchange that scans the message tracking logs to report on mail flow statistics.

Scripts can also use constructs such as **ForEach**, **If**, and **Switch** that are seldom used in a single command. You can use these constructs to process objects and make decisions in your scripts.

Windows PowerShell scripts have a .ps1 file extension. The most basic scripts are simply Windows PowerShell commands listed in a text file that has been saved with the .ps1 file extension. While the Windows PowerShell Integrated Scripting Environment (ISE) has better features, you can edit Windows PowerShell scripts by using a simple text editor such as Notepad.

Modifying scripts

Most administrators do not create their own scripts at first. Instead, they begin by using existing scripts and, if necessary, modifying those scripts to meet their needs. For example, you might start with a simple script that queries Active Directory Domain Services (AD DS) for users who have not signed in for an extended time and then generates a report. You could modify that script to automatically disable those stale accounts in addition to generating the report.

Generally, modifying an existing script is easier and faster than creating your own. The more complex the tasks, the more likely this is to be true. However, you should review all scripts and test them in a non-production environment before using them. Even if the author is not malicious, there could be a bug in the script, or you might be using the script in a way that the author did not intend.

PowerShell Gallery

Microsoft provides an organized set of scripts and modules in the PowerShell Gallery. The PowerShell Gallery contains content published by Microsoft and PowerShell Gallery members. You can use modules from the PowerShell gallery to simplify building your scripts.



Additional Reading: The PowerShell gallery is located at: <https://aka.ms/ue14hl>

Script Center repository

Microsoft also provides a set of scripts and code samples in the Script Center repository. Script Center is a website that provides instructional information for scripting on Windows servers and clients. The repository is the portion of that website with scripts and code samples. Many of the scripts and samples are provided by Microsoft, but there are also resources submitted from the community. This is a good source for code snippets that you can include in your scripts.

- Modifying an existing script is easier and faster than creating your own
- You should:
 - Understand how a downloaded script works
 - Test a downloaded script in a non-production environment
- You can get scripts from:
 - The PowerShell Gallery
 - The Script Center repository
 - Blogs and websites

Studijní materiály Okškolení



Additional Reading: The Script Center repository is located at: <https://aka.ms/g14v8f>

The Internet

There are many websites and blogs not associated with Microsoft that provide scripts and code samples. These resources are typically found by using a search engine with which you can search for how to complete a specific task by using Windows PowerShell. There are some very useful resources in the blogs and articles, but you should be very careful in your testing.

Creating scripts

If you cannot find a script that meets your needs, you can create your own script. You can still use resources such as the PowerShell Gallery and Script Center repository to find code snippets that are useful in building that script so you do not need to create all the code.

Scripting is very powerful. You can make changes to many objects, computers, or files very quickly. When you create and test scripts, it is likely that you will make mistakes. Some of those mistakes result in error messages. Other mistakes cause damage such as deleting objects or files. You should test your scripts in a development or test environment. Ideally, this environment should closely mimic your production environment. By testing in an isolated environment, you reduce the risk of causing damage with your scripts.

When you build your scripts, do so incrementally. Identify a logical order for the task you are trying to accomplish, and build the script in parts. This allows you to test each part during development and verify that it works as expected. For example, if you are creating a script to modify all users that are members of a group, the task will have two parts: a query that identifies the members of the group and a section that modifies the users. When you develop the syntax required to modify the users, you should modify a single user until you get the syntax correct.

- Create a script if you cannot find one that meets your needs
- Use code snippets from other sources when building your script
- Develop scripts in an isolated environment that cannot affect production
- Build scripts incrementally for easier testing during development

What is the **PowerShellGet** module?

The **PowerShellGet** module includes cmdlets for accessing and publishing items in the PowerShell Gallery. This module is part of the Windows Management Framework 5.0 that is included in Windows 10 and Windows Server 2016. You can upgrade other Windows operating systems to include Windows Management Framework 5.0 and thereby obtain the **PowerShellGet** module.

Alternatively, if you cannot update to Windows Management Framework 5.0, there is an .msi installer for **PowerShellGet** that can be used on systems with Windows PowerShell 3.0 or Windows PowerShell 4.0.

- Windows **PowerShellGet**:
- Has cmdlets for accessing and publishing items in the PowerShell Gallery
- Is included in Windows Management Framework 5.0
- Can be downloaded for Windows PowerShell 3.0 or Windows PowerShell 4.0
- Uses the NuGet provider
- You can implement a private PowerShell gallery
- Cmdlets for finding items are:
 - **Find-Module**
 - **Find-Script**

When you use the cmdlets in the **PowerShellGet** module for the first time, you are prompted to install the NuGet provider. NuGet is a package manager that can obtain and install packages on Windows. The cmdlets in the **PowerShellGet** module use the functionality in NuGet to interact with the PowerShell Gallery.

You can implement a private PowerShell gallery for your organization. You do this by creating your own NuGet feed. A NuGet feed can be created with a file share or a web-based application. When you have a private PowerShell gallery, you must register the NuGet feed by using the **Register-PSRepository** cmdlet and specify the source location. Once the repository is registered, it can be searched just like the PowerShell Gallery.

 **Additional Reading:** For more information about creating a NuGet feed, refer to: "Hosting your own NuGet feeds" at: <https://aka.ms/vm0ys1>

The following table lists the two cmdlets used most often to find cmdlets in the PowerShell Gallery.

Cmdlet	Description
Find-Module	This cmdlet is used to search for Windows PowerShell modules in the PowerShell Gallery. The simplest usage conducts searches based on the module name, but you can also search based on the command name, version, DscResource, and RoleCapability.
Find-Script	This cmdlet is used to search for Windows PowerShell scripts in the PowerShell Gallery. The simplest usage conducts searches based on the script name, but you can also search based on the version.

 **Additional Reading:** For more information about all of the PowerShellGet cmdlets, refer to: "PowerShellGet cmdlet reference" at: <https://aka.ms/tykgas>

Running scripts

Before you can begin modifying Windows PowerShell scripts or creating your own, you must know how to run a Windows PowerShell script. You might be familiar with the idea of double-clicking an executable to run it, but that process does not work for Windows PowerShell scripts.

One of the problems with many scripting languages is that running the scripts by accident is too easy. Users can accidentally run a script by double-clicking it. This is particularly a problem when the file extension is hidden and malware is included as an email attachment. For example, an attached file named receipt.txt.vbs would appear as receipt.txt and users would run it accidentally thinking it is a simple text file. This is not a concern for Windows PowerShell scripts because of the actions required to run a script.

- To enhance security, the .ps1 file extension is associated with Notepad
- Integration with File Explorer:
 - Open
 - Run with PowerShell
 - Edit
- To run scripts at the Windows PowerShell prompt:
 - Enter the full path - **C:\Scripts\MyScript.ps1**
 - Enter a relative path - **\Scripts\MyScript.ps1**
 - Reference the current directory - **.\MyScript.ps1**

Integration with File Explorer

To make Windows PowerShell scripts more secure, the .ps1 file extension is associated with Notepad. Therefore, when you double-click a .ps1 file, it opens in Notepad. This means that users can't be tricked into running a Windows PowerShell script by double-clicking it.

When you right-click a Windows PowerShell script, you have three options:

- **Open**. This option opens the script in Notepad.
- **Run with PowerShell**. This option runs the script, but the Windows PowerShell prompt does not remain open when the script completes.
- **Edit**. This option opens the script in the Windows PowerShell ISE.

In most cases, you want the Windows PowerShell prompt to remain open when you run a script. To do this, run the script from a Windows PowerShell prompt that is already open.

Running scripts at the Windows PowerShell prompt

When you run an executable file at a command prompt, you can enter the name of the executable to run the executable in the current directory. For example, when the current directory is **C:\app**, you can enter **app.exe** to run **C:\app\app.exe**. You cannot use this process to run Windows PowerShell scripts because it does not search the current directory.

To run a Windows PowerShell script at the Windows PowerShell prompt, you can use the following methods:

- Enter the full path to the script; for example, **C:\Scripts\MyScript.ps1**.
- Enter a relative path to the script; for example, **\Scripts\MyScript.ps1**.
- Reference the current directory; for example, **.\MyScript.ps1**.

The script execution policy

You can control whether Windows PowerShell scripts can be run on Windows computers. You do this by setting the execution policy on the computer. The default execution policy on a computer varies depending on the operating system version. To be sure of the current configuration, you can use the **Get-ExecutionPolicy** cmdlet.

The options for the execution policy are:

- **Restricted**. No scripts are allowed to be run.
- **AllSigned**. Scripts can be run only if they are digitally signed.
- **RemoteSigned**. Scripts that are downloaded can only be run if they are digitally signed.
- **Unrestricted**. All scripts can be run, but a confirmation prompt appears when running unsigned scripts that are downloaded.
- **Bypass**. All scripts are run without prompts.

- The options for the execution policy are:
 - **Restricted**
 - **AllSigned**
 - **RemoteSigned**
 - **Unrestricted**
 - **Bypass**
- Modify the execution policy by using:
 - **Set-ExecutionPolicy**
 - The **Turn on Script Execution** Group Policy setting
 - Override the execution policy for a single instance:
 - **Powershell.exe –ExecutionPolicy Bypass**
 - Remove downloaded status from a script by using **Unblock-File**



Note: Setting the script execution policy provides a safety net that can prevent untrusted scripts from being run accidentally. However, the execution policy can always be overridden.

You can set the execution policy on a computer by using the **Set-ExecutionPolicy** cmdlet. However, this is difficult to manage across many computers. When you configure the execution policy for many computers, you can use this Group Policy setting to override the local setting:

Computer Configuration\Policies\Administrative Templates\Windows Components\Windows PowerShell\Turn on Script Execution

You can override the execution policy for an individual Windows PowerShell instance. This is useful if company policy requires the execution policy to be set as **Restricted**, but you still must run scripts occasionally. To override the execution policy, run **PowerShell.exe** with the *-ExecutionPolicy* parameter:

```
Powershell.exe -ExecutionPolicy ByPass
```

If you have modified a downloaded script, the script will still have the attributes that identify it as a downloaded file. To remove that status from a script, use the **Unblock-File** cmdlet.

Demonstration: Setting the script execution policy

In this demonstration, you will see how to set the script execution policy.

Demonstration Steps

1. On **LON-CL1**, open **File Explorer**, and then browse to **E:\Mod08\Democode**.
2. Rename **HelloWorld.txt** to **HelloWorld.ps1**.
3. Verify what happens when you double-click **HelloWorld.ps1**.
4. Run **HelloWorld.ps1** by using the context menu.
5. Open a Windows PowerShell prompt.
6. To change the prompt location, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod08\Democode
```

7. To verify that **HelloWorld.ps1** is in the current directory, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ChildItem HelloWorld.ps1
```

8. To run **HelloWorld.ps1** by using the full path, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
E:\Mod08\Democode\HelloWorld.ps1
```

9. To verify that you cannot run **HelloWorld.ps1** without specifying a path, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
HelloWorld.ps1
```

10. To run **HelloWorld.ps1** in the current directory, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\HelloWorld.ps1
```

11. To view the current execution policy, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ExecutionPolicy
```

12. To prevent all scripts from running, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-ExecutionPolicy Restricted
```

13. To verify that all scripts are blocked, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\HelloWorld.ps1
```

14. To allow all scripts to be run, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-ExecutionPolicy Unrestricted
```

15. Leave the Windows PowerShell prompt open for the next demonstration.

Windows PowerShell and AppLocker

While the Windows PowerShell script execution policy provides a safety net for inexperienced users, it is not very flexible. When you set an execution policy, you can only check whether the script was downloaded and whether the script is signed.

Another alternative for controlling the use of Windows PowerShell scripts is AppLocker. With AppLocker, you can set various restrictions that limit execution of specific scripts or scripts in specific locations. Also, unlike the **AllSigned** execution policy, AppLocker can allow scripts that are signed only by specific publishers.

In Windows PowerShell 5.0, a new level of security has been added for using AppLocker to secure scripts. If a script is stopped at an interactive prompt for a purpose such as debugging, the commands entered at the interactive prompt can also be limited. When an AppLocker policy in Allow mode is detected, interactive prompts when running scripts are limited to ConstrainedLanguage mode.

ConstrainedLanguage mode allows all core Windows PowerShell functionality, such as scripting constructs. It also allows modules included in Windows to be loaded. However, it does limit access to running arbitrary code and accessing Microsoft .NET objects. ConstrainedLanguage mode blocks one of the vectors that an attacker could use to run unauthorized code.

- Running Windows PowerShell scripts can be controlled by using AppLocker
- AppLocker can limit scripts based on:
 - Name
 - Location
 - Publisher (via digital signature)
- In Windows PowerShell 5.0, interactive prompts are limited to ConstrainedLanguage mode



Additional Reading: For more information about ConstrainedLanguage mode, refer to: "about_Language_Modes in the Windows PowerShell help" or "About Language Modes" at: <https://aka.ms/nxcyid>

Digitally signing scripts

When you use Windows PowerShell scripts in your production environment, there should be an approval process to verify that those scripts have been tested. The approval process will vary depending on the size of the organization and the level of bureaucracy, but there should be some approval process.

One way to formalize the approval process for scripts used in a production environment is to digitally sign the scripts and use the **AllSigned** script execution policy. When you implement this, any modifications to scripts must be updated with a new digital signature. This prevents administrators or other staff from making script changes on the fly. For example, if your organization has a set of approved scripts for managing AD DS users, this configuration will prevent help desk staff from modifying the scripts either on purpose or by accident.

To add a digital signature to a script, you must have a code signing certificate that is trusted by all the computers that will be running the script. You can obtain a trusted code signing certificate from a public certification authority. You can also obtain a code signing certificate from an internal certification authority that is trusted by the computers.

You add a digital signature by using the **Set-AuthenticodeSignature** cmdlet as shown below:

```
$cert = Get-ChildItem -Path "Cert:\CurrentUser\My" -CodeSigningCert
Set-AuthenticodeSignature -FilePath "C:\Scripts\MyScript.ps1" -Certificate $cert
```

- Combine digital signatures on scripts with the **AllSigned** execution policy
- Use digitally signed scripts to:
 - Formalize the script approval process
 - Prevent accidental changes
- A trusted code signing certificate is required to add a digital signature to a script
- To set a digital signature:


```
$cert = Get-ChildItem -Path "Cert:\CurrentUser\My" -CodeSigningCert
Set-AuthenticodeSignature -FilePath "C:\Scripts\MyScript.ps1" -Certificate $cert
```

Demonstration: Digitally signing a script

In this demonstration, you will see how to digitally sign a Windows PowerShell script.

Demonstration Steps

Install a code signing certificate

1. On **LON-CL1**, open an **MMC** console, and then add the **Certificates** snap-in focused on **My user account**.
2. In the **MMC** console, browse to **Certificates - Current User\Personal**.
3. Use context menu of the **Personal** folder and select **Request New Certificate**.
4. Use the following settings in the **Certificate Enrollment** wizard:
 - o **Active Directory Enrollment Policy**
 - o **Adatum Code Signing template**

Studijní materiály pro školení

5. In the **MMC** console, verify that the new code signing certificate is present.
6. Close the **MMC** console.

Digitally sign a certificate

1. To view the code signing certificates installed for the current user, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
```

2. To place the code signing certificate in a variable, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$cert = Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
```

3. To digitally sign a script, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-AuthenticodeSignature -FilePath E:\Mod08\Democode\HelloWorld.ps1 -Certificate  
$cert
```

View the digital signature

- Open **D:\Allfiles\Mod08\Democode\HelloWorld.ps1**, and then verify that a digital signature has been added.

Lesson 2

Scripting constructs

While there are some simple scripts that use only simple Windows PowerShell commands, most scripts use scripting constructs to perform more complex actions. You can use the **ForEach** construct to process all of the objects in an array. You can use **If..Else** and **Switch** constructs to make decisions in your scripts. Finally, there are less common looping constructs such as **For**, **While**, and **Do..While** loops that can be used when appropriate. In this lesson, you will learn how to use scripting constructs.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the syntax of the **ForEach** construct.
- Explain how to use the **ForEach** construct.
- Describe the syntax of the **If** construct.
- Explain how to use the **If** construct.
- Describe the syntax of the **Switch** construct.
- Explain how to use the **Switch** construct.
- Describe how to use the **For** construct.
- List the other loop constructs.
- Explain how to use **Break** and **Continue**.

Understanding ForEach loops

When you perform piping, the commands in the pipeline are applied to each object. In some cases, you might need to use the **ForEach-Object** cmdlet to process the data in the pipeline. When you store data in an array, the **ForEach** construct allows you to process each item in the array.

The **ForEach** construct uses the following syntax:

- Example:
- ```
ForEach ($user in $users) {
 Set-ADUser $user -Department "Marketing"
}
```
- Commands between braces are processed once for each item in the array
  - You do not need to know how many items are in the array
  - \$user contains each array item during the loop
  - The indent is to make it easier to read
  - Variable names should be meaningful

```
ForEach ($user in $users) {
 Set-ADUser $user -Department "Marketing"
}
```

In the example above, there is an array named `$users` that contains AD DS user objects. The **ForEach** construct processes the Windows PowerShell commands between the braces once for each object. When the commands are being processed, `$user` is the variable that contains each item from the array. On the first iteration, `$user` contains `$users[0]`, and on the second iteration, `$user` contains `$users[1]`. This continues until all items in the array have been processed once.

Studijní materiály Okškolení

In a script, the **ForEach** construct is the most common way to process items that you have placed into an array. It is easy to use because you do not need to know the number of items to process them.

The example above has only one command between the braces, but you can add many commands, which will be processed for each loop. The indent of commands between the braces is by convention to make the script easier to read. The indent is not a technical requirement, but it is a good practice.

Naming of variables in the **ForEach** loop should be meaningful. Most of the time, you clearly identify the variable used in the loop as a single instance of the array. For example, for an array named \$users, the variable used in the loop could be \$user. You might see examples of variables with a single letter that is the same as the initial letter of the array. However, this should only be used in simple code where it is easy to tell that they are related.

## Demonstration: Using a ForEach loop

In this demonstration, you will see how to use the **ForEach** construct.

### Demonstration Steps

1. Open E:\Mod08\Democode\10961C\_Mod08\_Demo3.ps1 in Windows PowerShell ISE.
2. Review the code, and then run the script.

## Understanding the If construct

You can use the **If** construct in Windows PowerShell to make decisions. You also can use it to evaluate data you have queried or user input. For example, you could have an **if** statement that displays a warning if available disk space is low.

The **If** construct uses the following syntax:

- Use the **If** construct to make decisions
  - There can be zero or more **ElseIf** statements
  - **Else** is optional
  - Example:
- ```
If ($freeSpace -le 5GB) {
    Write-Host "Free disk space is less than 5 GB"
} ElseIf ($freeSpace -le 10GB) {
    Write-Host "Free disk space is less than 10 GB"
} Else {
    Write-Host "Free disk space is more than 10 GB"
}
```

```
If ($freeSpace -le 5GB) {
    Write-Host "Free disk space is less than 5 GB"
} ElseIf ($freeSpace -le 10GB) {
    Write-Host "Free disk space is less than 10 GB"
} Else {
    Write-Host "Free disk space is more than 10 GB"
}
```

In the example above, the condition **\$freeSpace -le 5GB** is evaluated first. If this condition is true, the script block in the braces is executed, and no further processing happens in the **If** construct. In this case, a message indicating there is less than 5 GB of free disk space is displayed.

If the first condition is not true, the condition **\$freeSpace -le 10GB** that is defined for **ElseIf** is evaluated. If this condition is true, the script block in the braces is executed, and no further processing happens in the **If** construct. In this example, there is a single **ElseIf**, but you can have multiple **ElseIf** statements or none.

If all conditions are not true, then the script block for **Else** is executed. **Else** is optional.

 **Note:** When you are making multiple decisions based on a single variable, using multiple **ElseIf** script blocks rather than nesting multiple **If** statements is preferred.

Demonstration: Using the If construct

In this demonstration, you will see how to use the **If** construct.

Demonstration Steps

1. Open **E:\Mod08\Democode\10961C_Mod08_Demo3.ps1** in Windows PowerShell ISE.
2. Review the code, and then run the script.
3. Update the value of **\$freeSpace** to **11GB**, and then run the script again.
4. Update the value of **\$freeSpace** to **22GB**, and then run the script again.

Understanding the Switch construct

The **Switch** construct is similar to an **If** construct that has multiple **ElseIf** sections. The **Switch** construct evaluates a single variable or item against multiple values and a script block for each value. The script block for each value is executed if that value matches the variable. There is also a **Default** section that executes only if there are no matches.

The **Switch** construct uses the following syntax:

- The **Switch** construct compares a variable to a list of values
- Example:


```
Switch ($choice) {
    1 { Write-Host "You selected menu item 1" }
    2 { Write-Host "You selected menu item 2" }
    3 { Write-Host "You selected menu item 3" }
    Default { Write-Host "You did not select a valid option" }
}
```
- You can also use wildcards and regular expressions
- Multiple matches are possible

```
Switch ($choice) {
    1 { Write-Host "You selected menu item 1" }
    2 { Write-Host "You selected menu item 2" }
    3 { Write-Host "You selected menu item 3" }
    Default { Write-Host "You did not select a valid option" }
}
```

In addition to matching values, the **Switch** construct can also be used to match patterns. You can use the **-wildcard** parameter to perform pattern matching by using the same syntax as the **-like** operator. Alternatively, you can use the **-regex** parameter to perform matching by using regular expressions.

It is important to be aware that when you use pattern matching, multiple matches are possible. When there are multiple matches, the script blocks for all matching patterns are executed. This is different from an **If** construct in which only one script block is executed.

The following example uses pattern matching:

```
Switch -WildCard ($ip) {
    "10.*" { Write-Host "This computer is on the internal network" }
    "10.1.*" { Write-Host "This computer is in London" }
    "10.2.*" { Write-Host "This computer is in Vancouver" }
    Default { Write-Host "This computer is not on the internal network" }
```

For values of \$ip on the London or Vancouver networks, two messages will be displayed. If \$ip includes an IP address on the 10.x.x.x network, the messages will indicate that the computer is on the internal network and that the computer is in either London or Vancouver. If \$ip is not an IP address on the 10.x.x.x network, the message indicates it is not on the internal network.

If you provide multiple values in an array to a **Switch** construct, each item in the array is evaluated. In the example above, if the variable \$ip was an array with two IP addresses, then both IP addresses would be processed. The actions appropriate for each item in the array would be performed.

Demonstration: Using the Switch construct

In this demonstration, you will see how to use the **Switch** construct.

Demonstration Steps

1. Open E:\Mod08\Democode\10961C_Mod08_Demo3.ps1 in Windows PowerShell ISE.
2. Review the code, and then run the script.
3. Update the value of \$computer to **VAN-SRV1**, and then run the script again.
4. Update the value of \$computer to **SEA-CL1**, and then run the script again.
5. Update the value of \$computer to **SEA-RDP**, and then run the script again.

Understanding the For construct

The **For** construct performs a series of loops similar to a **ForEach** construct. However, when using the **For** construct, you must define how many loops occur, which is useful when you want an action to be performed a specific number of times. For example, you could create a specific number of user accounts in a test environment.

The **For** construct uses the following syntax:

- The **For** construct is used to run a script block a specific number of times based on the:
 - Initial state
 - Condition
 - Action
- Example:


```
For($i=1; $i -le 10; $i++) {
    Write-Host "Creating User $i"
}
```

```
For($i=1; $i -le 10; $i++) {
    Write-Host "Creating User $i"
}
```

The **For** construct uses an initial state, a condition, and an action. In the example above, the initial state is **\$i=1**. The condition is **\$i -le 10**. When the condition specified is true, another loop is processed. After each loop is processed, the action is performed. In this example, the action is **\$i++** which increments **\$i** by 1.

The script block inside the braces is run each time the loop is processed. In the example above, this loop is processed 10 times.



Note: When you are processing an array of objects, using the **ForEach** construct is preferred because you do not need to calculate the number of items in the array before processing.

Understanding other loop constructs

There are other less common looping constructs that you can use when required. These are

Do..While, **Do..Until**, and **While**. All of these process a script block until a condition is met, but they vary in the details of how they do it.

Do..While

The **Do..While** construct executes a script block until a specified condition is not true. This construct guarantees that the script block is executed at least once.

The **Do..While** construct uses the following syntax:

```
Do {
    Write-Host "Script block to process"
} While ($answer -eq "go")
```

• Do..While

- Loops until the condition is false
 - Guaranteed to process the script block once
- ```
Do {
 Write-Host "Code block to process"
} While ($answer -eq "go")
```

#### • Do..Until

- Loops until the condition is true
  - Guaranteed to process the script block once
- ```
Do {
    Write-Host "Code block to process"
} Until ($answer -eq "stop")
```



Do..Until

The **Do..Until** construct executes a script block until a specified condition is true. This construct guarantees that the script block is executed at least once.

The **Do..Until** construct uses the following syntax:

```
Do {
    Write-Host "Script block to process"
} Until ($answer -eq "stop")
```

While

The **While** construct executes a script block until a specified condition is false. It is similar to the **Do..While** construct, except that the script block is not guaranteed to be executed.

The **While** construct uses the following syntax:

```
While ($answer -eq "go") {
    Write-Host "Script block to process"
}
```

Studijní materiály Okškolení

Understanding Break and Continue

Break and **Continue** are two commands that you can use to modify the default behavior of a loop. **Continue** ends the processing for the current iteration of the loop. **Break** completely stops the loop processing. You typically use these commands when the data you are processing has an invalid value.

In this example, the use of **Continue** prevents modification of the Administrator user account in the list of users to be modified:

```
ForEach ($user in $users) {
    If ($user.Name -eq "Administrator") {Continue}
    Write-Host "Modify user object"
}
```

In this example, **Break** is used to end the loop when a maximum number of accounts has been modified:

```
ForEach ($user in $users) {
    $number++
    Write-Host "Modify User object $number"
    If ($number -ge $max) {Break}
}
```

- **Continue** stops processing the current iteration of a loop


```
ForEach ($user in $users) {
        If ($user.Name -eq "Administrator") {Continue}
        Write-Host "Modify user object"
    }
```
- **Break** completely stops loop processing


```
ForEach ($user in $users) {
        $number++
        Write-Host "Modify User object $number"
        If ($number -ge $max) {Break}
    }
```

Lesson 3

Importing data from files

When you create a script, reusing data from other sources is useful. Most applications can export the data you want in various formats such as a CSV file or an XML file. You can use Windows PowerShell cmdlets to import data in different formats for use in your scripts. In this lesson, you will learn how to import data from a text file, CSV file, XML file, and JavaScript Object Notation (JSON).

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to use **Get-Content** to read file data.
- Describe how to use **Import-Csv** to retrieve data.
- Describe how to use **Import-CliXML** to import XML data.
- Describe how to use **ConvertFrom-Json** to work with JSON data.
- Explain how to import data from text, CSV, and XML files.

Using Get-Content

You can use **Get-Content** to retrieve data from a text file for use in your scripts. The information retrieved from the text file is stored in an array; each line in the text file is an item in the array.

The typical syntax for Get-Content is:

- **Get-Content** retrieves content from a text file
 - Each line in the file becomes an item in an array
 - \$computers = Get-Content "C:\Scripts\computers.txt"
- Import multiple files by using wildcards
 - Get-Content -Path "C:\Scripts*" -Include "*.txt","*.log"
- Limit the data retrieved by using the parameters:
 - -TotalCount
 - -Tail

```
$computers = Get-Content C:\Scripts\computers.txt
```

The example above retrieves a list of computer names from the **computers.txt** file. The **\$computers** variable stores each of the computer names and can be processed. For example, you could use a **ForEach** construct to do some processing on each computer in the list. Over time, as the list of computers changes, the script automatically picks them up from the **computers.txt** file.

You can use wildcards in the path for **Get-Content** to obtain data from multiple files at a time. When you use wildcards for the path, you can modify the files selected by using the **-Include** and **-Exclude** parameters. When you use **-Include**, only the specified patterns are included. When you use **-Exclude**, all files are included except the patterns specified. Using wildcards can be useful when you want to scan all text files for specific content such as an error in log files.

The syntax for using **-Include** is:

```
Get-Content -Path "C:\Scripts\*" -Include "*.txt","*.log"
```

You can limit the amount of data that you retrieve with **Get-Content** by using the *-TotalCount* and *-Tail* parameters. The *-TotalCount* parameter specifies how many lines should be retrieved from the beginning of a file. The *-Tail* parameter specifies how many lines to retrieve from the end of a file. For example:

```
Get-Content C:\Scripts\computers.txt -TotalCount 10
```

Using Import-Csv

Many applications can export data to a CSV file. This ability makes the **Import-Csv** cmdlet very useful because it can import data that was exported from those applications. When the CSV file is imported, each line in the file, except the first row, becomes an item in an array. The first row in the CSV file is a header row that is used to name the properties of each item in the array.

The **Import-Csv** cmdlet uses the following syntax:

- The first row in the CSV file is a header row
- Each line in the CSV file becomes an array item
- The header row defines the property names for the items
- `$users = Import-Csv C:\Scripts\Users.csv`
- You can specify a custom delimiter by using the *-Delimiter* parameter
- You can specify a missing header row by using the *-Header* parameter

```
$users = Import-Csv C:\Scripts\Users.csv
```

Example data for **Users.csv**:

```
First,Last,UserID,Department
Amelie,Garner,AGarner,Sales
Evan,Norman,ENorman,Sales
Siu,Robben,SRobben,Sales
```

When you run the example above, the data from **Users.csv** is placed in the `$users` array. There are three items in the array. Each item in the array has four properties named in the header row. You can reference each of the properties by name. For example:

```
$users[2].UserID
```

Some programs export data using a delimiter other than a comma. If your data uses an alternate delimiter, you can specify which character by using the *-Delimiter* parameter.

If your data file does not include a header row, you can provider names for the columns by using the *-Header* parameter. You can provide a list of property names in the command or provide an array that contains the property names. When you use the *-Header* parameter, all rows in the file become items in the imported array.

Using Import-Clixml

XML is a more complex data storage format than CSV files. The main advantage of using XML for Windows PowerShell is that it can hold multiple levels of data. A CSV file works with a table of information in which the columns are the object properties. In a CSV file, it is difficult to work with multivalued attributes, while XML can easily represent multivalued attributes or even objects that have other objects as a property.

Using **Import-Clixml** to retrieve data from an XML file creates an array of objects. Because XML can be complex, you might not easily be able to understand the object properties by reviewing the contents of the XML file directly. You can use **Get-Member** to identify the properties of the data that you import.

The **Import-Clixml** cmdlet uses the following syntax:

```
$users = Import-Clixml C:\Scripts\Users.xml
```

You can limit the data retrieved by **Import-Clixml** by using the **-First** and **-Skip** parameters. The **-First** parameter specifies to retrieve only the specified number of objects from the beginning of the XML file. The **-Skip** parameter specifies to ignore the specified number of objects from the beginning of the XML file and retrieve all the remaining objects.

- XML can store more complex data than CSV files
- **Import-Clixml** creates an array of objects
- \$users = Import-Clixml C:\Scripts\Users.xml
- Use **Get-Member** to view object properties
- Limit the data retrieved by using the parameters:
 - -First
 - -Skip

Using ConvertFrom-Json

JSON is a lightweight data format that is similar to XML. JSON is lightweight compared to XML due to its simpler syntax. JSON is similar to XML because it can represent multiple layers of data.

Windows PowerShell does not include cmdlets that import or export JSON data directly from a file. Instead, if you have JSON data stored in a file, you can retrieve the data by using **Get-Content** and then convert the data by using the **ConvertFrom-Json** cmdlet.

The **ConvertFrom-Json** cmdlet uses the following syntax:

```
$users = Get-Content C:\Scripts\Users.json | ConvertFrom-Json
```

- JSON is:
 - A lightweight data format similar to XML
 - Commonly used by web services
- You can convert from JSON, but not import directly
- \$users = Get-Content C:\Scripts\Users.json | ConvertFrom-Json
- Retrieve JSON data directly from web services by using **Invoke-RestMethod**
- \$users = Invoke-RestMethod "https://hr.adatum.com/api/staff"

Invoke-RestMethod

JSON is a format commonly used by web services to provide data. You can query data directly from a web service by using **Invoke-RestMethod**. **Invoke-RestMethod** sends a request to the specified URL and obtains data from the response. The retrieved data is automatically converted to objects. You do not need to use **ConvertFrom-Json**.

The **Invoke-RestMethod** cmdlet uses the following syntax:

```
$users = Invoke-RestMethod "https://hr.adatum.com/api/staff"
```

 **Note:** The URLs used to retrieve data from a web service are not standardized. You must review the documentation for the web service to identify the correct URLs to retrieve data.

 **Note:** **Invoke-RestMethod** is also capable of working with XML, RSS feeds, and ATOM feeds.

Demonstration: Importing data

In this demonstration, you will see how to import data.

Demonstration Steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. To retrieve data from a text file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Content E:\Mod08\Democode\computers.txt
```

3. To place data from a text file into an array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computers = Get-Content E:\Mod08\Democode\computers.txt
```

4. To display the number of items in the `$computers` array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computers.Count
```

5. To display the items in the `$computers` array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computers
```

6. To import CSV data, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Import-Csv E:\Mod08\Democode\users.csv
```

7. To import CSV data into an array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$users = Import-Csv E:\Mod08\Democode\users.csv
```

8. To display the count of items in the array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$users.Count
```

9. To display the first item in the array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$users[0]
```

10. To display the property named **First** for the item, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$users[0].First
```

11. To import data from an XML file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Import-Clixml E:\Mod08\Democode\users.xml
```

12. To import XML data into an array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$usersXml = Import-Clixml E:\Mod08\Democode\users.xml
```

13. To view the number of items in the array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$usersXml.Count
```

14. To view the first item in the array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$usersXml[0]
```

15. To view the properties for the items in the array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$usersXml | Get-Member
```

16. Close the Windows PowerShell prompt.

Studijní materiály Okskolení

Lab: Basic scripting

Scenario

You have started to develop Windows PowerShell scripts to simplify administration in your organization. There are multiple tasks to accomplish, and you will write a Windows PowerShell script for each one.

Objectives

After completing this lab, you will be able to:

- Digitally sign a script.
- Process an array by using **ForEach**.
- Process items by using **If** statements.
- Create a random password.
- Create user accounts based on a CSV file.

Lab Setup

Estimated Time: **120 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before beginning the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Exercise 1: Signing a script

Scenario

In this exercise, you will learn how to modify the script execution policy and digitally sign a script.

The main tasks for this exercise are as follows:

1. Install a code signing certificate.
2. Digitally sign a certificate.
3. Set the execution policy.

► **Task 1: Install a code signing certificate**

1. On **LON-CL1**, open an **MMC** console, and then add the **Certificates** snap-in focused on **My user account**.
2. In the **MMC** console, browse to **Certificates - Current User\Personal**.
3. Use the context menu of the **Personal** folder and select **Request New Certificate**.
4. Use the following settings in the **Certificate Enrollment** wizard:
 - **Active Directory Enrollment Policy**
 - **Adatum Code Signing** template
5. In the **MMC** console, verify that the new code signing certificate is present.
6. Close the **MMC** console.

► **Task 2: Digitally sign a certificate**

1. Open a **Windows PowerShell** prompt.
2. Place the code signing certificate in **Cert:\CurrentUser\My** into a variable.
3. In **E:\Mod08\Labfiles**, rename **HelloWorld.txt** to **HelloWorld.ps1**.
4. Use the **Set-Authenticode** cmdlet to apply a digital signature to **HelloWorld.ps1**.

► **Task 3: Set the execution policy**

1. On **LON-CL1**, set the execution policy to allow only signed scripts.
2. Verify that you can run **HelloWorld.ps1**.
3. Set the execution policy to **Unrestricted**.

Results: After completing this exercise, you should have digitally signed a script and modified the script execution policy.

Exercise 2: Processing an array with a **ForEach** loop

Scenario

A. Datum is testing a new voice over IP (VoIP) and video conferencing system. To support this system, you must set the *ipPhone* attribute for a group of test users. The naming convention that has been selected for the *ipPhone* attribute is **FirstName.LastName@adatum.com**.

The main tasks for this exercise are as follows:

1. Create a test group.
2. Create a script to configure the *ipPhone* attribute.

► Task 1: Create a test group

1. On **LON-CL1**, use PowerShell to create a new AD DS group named **IPPhoneTest** in the **IT** organizational unit.
2. Add the following users as members in the **IPPhoneTest** group:
 - **Abbi Skinner**
 - **Ida Alksne**
 - **Parsa Schoonen**
 - **Tonia Guthrie**

► Task 2: Create a script to configure the ipPhone attribute

1. Create a script named **E:\Mod08\Labfiles\ipPhone.ps1**, and then open it in the Windows PowerShell ISE.
2. Use **Get-ADGroupMember** to create a query to obtain the membership of the **IPPhoneTest** group.
3. Create a **ForEach** loop that processes the users that are members of **IPPhoneTest**.
4. In the loop:
 - Use **Get-ADUser** to retrieve the first and last names for a user.
 - Calculate the required value for the **ipPhone** attribute.
 - Use **Set-ADUser** with the **-Replace** parameter to set the **ipPhone** attribute for the user.
5. Run the script and then verify that the **ipPhone** attribute is modified for the selected users.

Results: After completing this exercise, you should have configured an Active Directory attribute by using a **ForEach** loop.

Exercise 3: Processing items by using If statements

Scenario

Some of the servers in your organization have services that do not start properly when the server is restarted. You want to create a script that can be used to start a specified list of services. When you have performed sufficient testing, you plan to configure a scheduled task that runs the script. During the testing phase, you will work with the Windows Time and Print Spooler services.

The main tasks for this exercise are as follows:

1. Create services.txt with service names.
2. Create a script that starts stopped services.

► Task 1: Create services.txt with service names

1. On **LON-CL1**, open **Windows PowerShell**.
2. Create a new file **E:\Mod08\Labfiles\services.txt**.
3. Identify the correct name for the **Print Spooler** service and add it to **services.txt**.
4. Identify the correct name for the **Windows Time** service and add it to **services.txt**.

► **Task 2: Create a script that starts stopped services**

1. Create a new script **E:\Mod08\Labfiles\StartServices.ps1**.
2. Retrieve the service names from **services.txt** and place them in a variable.
3. Use a **ForEach** loop to process each service:
 - o If the service is not running, start it and write text to the screen indicating that the service was started.
 - o If the service is running, do nothing and write text to the screen indicating that no action was required.

Results: After completing this exercise, you should have created a script that starts a list of services that are defined in a text file.

Exercise 4: Creating a random password

Scenario

In this exercise, you will use a **For** loop to create a password composed of a specified number of random characters and write it to the screen. The length of the password generated can be modified by updating the value of a variable at the start of the script. You must generate random numbers and convert them to characters.

The main task for this exercise is as follows:

1. Create a script that generates random passwords.

► **Task 1: Create a script that generates random passwords**

1. Create a new script **E:\Mod08\Labfiles\CreatePasswords.ps1**.
2. At the start of the script, set the variable **\$passwordLength** to **8**.
3. Use the **For** construct to create a loop that runs the number of times specified by **\$passwordLength**.
 - o Generate a random number with a **minimum** value of **65** and **maximum** value of **90**.
 - o Generate a letter from the random number by converting it to be a **[char]** variable.
 - o Add the letter to the **\$password** variable.
4. Display the password on the screen.

Results: After completing this exercise, you should have created a script that generates a random password.

Exercise 5: Creating users based on a CSV file

Scenario

In this exercise, you will import a CSV file and use the data to create user accounts in AD DS.

The main tasks for this exercise are as follows:

1. Create AD DS users from a CSV file.
2. Prepare for the next module.

► Task 1: Create AD DS users from a CSV file

1. Create a new script named **E:\Mod08\Labfiles\CreateUsers.ps1**.
2. Import **users.csv** and store the objects in a variable.
3. Create a **ForEach** loop that processes the data in the variable to create user accounts:
 - o Create a variable that contains the LDAP name of the organizational unit for the user. For example: OU=IT,DC=Adatum,DC=com
 - o Create a variable that contains the user principal name for the new user. This should be in the format **UserID@adatum.com**.
 - o Create a variable that contains the display name for the new user. This should be in the format of *FirstName LastName*.
 - o Write a status message to the screen indicating which user is being created and where.
 - o Create the new user and be sure to set the following:
 - **GivenName**
 - **Surname**
 - **Name**
 - **DisplayName**
 - **SamAccountName**
 - **UserPrincipalName**
 - **Path**
 - **Department**

Results: After completing this exercise, you should have created users based on the data in a CSV file.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: When would you prefer to use a code signing certificate from a third-party certification authority rather than an internal certification authority?

Question: In Exercise 2, you configured the **ipPhone** attribute for a group of test users. How would you update that script for a larger set of users as the solution is deployed to the rest of the organization?

Studijní materiály Okškolení

Module Review and Takeaways

Review Questions

Question: When importing data, what is the primary consideration when selecting which cmdlet to use?

Question: Why is the **ForEach** construct used more often than the **For** construct?

Module 9

Advanced scripting

Contents:

Module Overview	9-1
Lesson 1: Accepting user input	9-2
Lesson 2: Overview of script documentation	9-9
Lab A: Accepting data from users	9-13
Lesson 3: Troubleshooting and error handling	9-16
Lesson 4: Functions and modules	9-24
Lab B: Implementing functions and modules	9-30
Module Review and Takeaways	9-33

Module Overview

In this module, you will learn more advanced techniques that you can use in scripts. These techniques include gathering user input, reading input from files, documenting scripts with help information, and handling errors. These techniques help you make scripts that are more useful and user friendly.

Objectives

After completing this module, you will be able to:

- Accept user input for a script.
- Explain script documentation.
- Implement error handling for a script.
- Explain functions and modules.

Lesson 1

Accepting user input

To enhance the usability of your scripts, you must learn how to accept user input. This skill allows you to create scripts that can be used for multiple purposes. In addition, accepting user input allows you to create scripts that are easy for others to use. In this lesson, you learn about multiple methods for accepting user input in a script.

Lesson Objectives

After completing this lesson, you will be able to:

- Identify values in a script that are likely to change.
- Explain how to use **Read-Host** to accept user input.
- Explain how to use **Get-Credential** to accept user credentials.
- Explain how to use **Out-GridView** to obtain user input.
- Obtain user input by using **Read-Host**, **Get-Credential**, and **Out-GridView**.
- Explain how to pass parameters to a script.
- Obtain user input by using parameters.

Identifying values that might change

When you first write a script, it is usually done to meet a need at a specific point in time. For example, you might need to find all the Active Directory Domain Services (AD DS) user accounts that have not signed in to the domain for 60 days. Or, you might need to determine which of your domain controllers have a specific event in the event logs in the previous 30 days.

Over time, you likely will find that you need to use variations of your scripts. For example, you might need to find computer accounts that have not signed in for 30 days. Or, you might need to find specific events in the event logs of servers other than your domain controllers.

In the examples above, there are items in the script that are changing. The first and simplest way to address this is to put values that are likely to change in a variable. By placing that variable at the top of the script where it is easily accessible, you make it easier to modify the script. However, this still requires that the script be modified.

In an environment where many administrators share a common set of scripts, it is better not to modify scripts that will go through an approval process. If scripts are digitally signed, each modification requires that the script be signed again. It is preferable to accept user input for values that change rather than modify scripts.

- Scripts might initially have static values:
 - Find users that have not signed in for 30 days
 - Find specific events on domain controllers
- When scripts are reused, some of those values might need to change
 - To simplify changing values in scripts:
 - Use variables defined at the top of the script
 - To avoid changing scripts:
 - Accept user input

Using Read-Host

You can use the **Read-Host** cmdlet to obtain input from users while a script is executing. The request for user input could be a prompt to start the script or based-on results from processing that has already happened in the script. For example, after performing a query for AD DS user objects and displaying the number of objects retrieved, the script could prompt a decision on whether to continue or stop. Or, the script could request the specific event ID for which to search. The syntax for the **Read-Host** cmdlet is shown below:

```
$answer = Read-Host "How many days"
```

The example above will stop processing the script and prompt the user with text as shown below:

```
How many days:
```

At the prompt, the user types in a response and presses Enter. The response that the user provides is placed in the variable \$answer.

When you display text as part of using **Read-Host**, a colon (:) is always appended to the end of the text. There is no parameter to suppress the behavior. However, if you use **Read-Host** without displaying text, no colon is displayed. You can combine a **Write-Host** command with **Read-Host** to display text and avoid a colon being appended, as the following shows:

```
Write-Host "How many days? " -NoNewline
$answer = Read-Host
```

- Use **Read-Host** to request user input while a script is running:

```
$answer = Read-Host "How many days?"
```

- To avoid displaying a colon, combine **Write-Host** and **Read-Host**:

```
Write-Host "How many days?" -NoNewline
```

```
$answer = Read-Host
```

Using Get-Credential

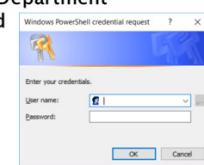
As a best practice, administrators should have two user accounts. Each administrator should have a standard user account that is used for day-to-day activity and a second account with administrative permissions. Separating these roles helps to avoid accidental damage to computer systems and limits the potential effects of malware. The **Get-Credential** cmdlet can help you use the administrative account while still signed in as a standard user account.

Many scripts that administrators run require elevated privileges. For example, a script that creates AD DS user accounts requires administrative privileges. Even querying event logs from a remote computer might require administrative privileges.

One way to elevate privileges when you run a script is to use the **Run as administrator** option when you open the Windows PowerShell prompt. If you use **Run as administrator**, you are prompted for credentials, and all actions performed at that Windows PowerShell prompt use the credentials provided.

- You can request credentials and use them to run commands in a script:

```
$cred = Get-Credential
Set-ADUser -Identity $user -Department
"Marketing" -Credential $cred
```



- You can customize the credential prompt by using parameters:

- -Message
- -UserName

- You can store credentials securely in a file:

```
$cred | Export-Clixml C:\cred.xml
```

As an alternative to using **Run as administrator** for running a script, you can have your script prompt for credentials instead. Many Windows PowerShell cmdlets allow an alternate set of credentials to be provided. That way, the credentials that the script obtains can be used to run individual commands in the script. You can prompt for credentials by using **Get-Credential**. The syntax for using the **Get-Credential** cmdlets is shown below:

```
$cred = Get-Credential
Set-ADUser -Identity $user -Department "Marketing" -Credential $cred
```

The default text in the pop-up window is "Enter your credentials." You can customize this text to be more descriptive by using the *-Message* parameter. You can also fill in the **User name** box by using the *-UserName* parameter.

Storing credentials

You can store credentials to file for later reuse without being prompted for credentials. To store credentials to file, you use **Export-Clixml**. For a credential object, **Export-Clixml** encrypts the credential object before storing it in an XML file. The syntax to store a credential object to file is the following:

```
$cred | Export-Clixml C:\cred.xml
```

The encryption used by **Export-Clixml** is user-specific and computer specific. That means that if you store the encrypted credentials, only you can retrieve the encrypted credentials and only on the computer you originally used to store them. This keeps the credentials secure, but it also means that they cannot be shared with other users.

Using Out-GridView

Out-GridView is primarily used to view data. However, you can also use **Out-GridView** to create a simple menu selection interface. When the user makes one or more selections in the window presented by **Out-GridView**, the data for those objects is either passed further through the pipeline or placed into a variable. The syntax for selecting an option in **Out-GridView** is shown below:

```
$selection = $users | Out-GridView -PassThru
```

- **Out-GridView** can be used as a simple menu system
`$selection = $users | Out-GridView -PassThru`
- For more control over the items selected, you can use the *-OutputMode* parameter.

Value	Description
None	Do not allow any rows to be selected
Single	Allows zero rows or one row to be selected
Multiple	Allows zero rows, one row, or multiple rows to be selected

In the example above, an array of user accounts is piped to **Out-GridView**. **Out-GridView** displays the user accounts on screen, and the user can select one or more rows in the **Out-GridView** window. When the user clicks **OK**, the selected rows are stored in the `$selection` variable. You could then perform further processing on the users' accounts.

To retain more control over the amount of data that users can select, you can use the `-OutputMode` parameter instead of the `-PassThru` parameter. The following table shows the values that can be defined for the `-OutputMode` parameter.

Value	Description
None	This is the default value that does not pass any objects further down the pipeline.
Single	This value allows users to select zero rows or one row in the Out-GridView window.
Multiple	This value allows users to select zero rows, one row, or multiple rows in the Out-GridView window. This value is equivalent to using the <code>-PassThru</code> parameter.

 **Note:** Because users are not forced to select a row in the **Out-GridView** window, you must ensure that your script properly handles the scenario where a row is not selected.

Demonstration: Obtaining user input

In this demonstration, you will see how to obtain user input.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. To obtain user input by using **Read-Host**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$days = Read-Host "Enter the number of days"
```

3. To view the data obtained by **Read-Host**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$days
```

4. To obtain a credential, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$cred = Get-Credential
```

5. To display the credential information, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$cred | Format-List
```

6. To store the credential in a file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$cred | Export-Clixml -Path E:\Mod09\Democode\cred.xml
```

7. To view the content of the file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Content E:\Mod09\Democode\cred.xml
```

8. To display a list of computer accounts in **Out-GridView**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ADComputer -Filter * | Out-GridView
```

9. To allow a single section in the **Out-GridView** window, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computer = Get-ADComputer -Filter * | Out-GridView -OutputMode Single
```

10. Select **LON-CL1** from the list.

11. To display the selected object, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$computer
```

12. Close the Windows PowerShell prompt.

Passing parameters to a script

You can configure your scripts to accept parameters in the same way that cmdlets do. This is a good method for users to provide input because it is consistent in how users provide input for cmdlets. The consistency makes it easy for users to understand.

To identify the variables that will store parameter values, you use a **Param()** block. The variable names are defined between the parentheses. The syntax for using a **Param()** block is shown below:

```
Param(  
    [string]$ComputerName  
    [int]$EventID  
)
```

- Use a **Param()** block to identify the variables that will store parameter values:

```
Param(  
    [string]$ComputerName  
    [int]$EventID  
)
```

- The variable names in the **Param()** block define the parameter names:

```
.\GetEvent.ps1 -ComputerName LON-DC1  
-EventID 5772
```

The variable names defined in the **Param()** block are also the names of the parameters. In the example above, the script containing this **Param()** block has the **-ComputerName** and **-EventID** parameters that can be used. When you type the parameter names for the script, you can use tab completion just as you can for cmdlet parameters. The syntax for executing a script with parameter is shown below:

```
.\GetEvent.ps1 -ComputerName LON-DC1 -EventID 5772
```



Note: Parameters are positional by default. If the parameter names are not specified, then the parameter values are passed to the parameters in order. For example, the first value after the script name is placed in the first parameter variable.



Note: If you do not put a **Param()** block in your script, you can still pass data into the script by using unnamed parameters. The values that are provided after the script name are available inside the script in the **\$args** array.

Defining variable types

It is a best practice to define variable types in a **Param()** block. When you define the variable types, if a user enters a value that cannot be converted to that variable type, an error is generated. This is one method for you to validate the data that users enter.

You can use the switch variable type for a parameter when there is an option that you want to be on or off. When the script is executed, the parameter's presence sets the variable to **\$true**. If the parameter is not present then the value for a variable is **\$false**. For example, in a script that typically displays some status information to users, you could create a **-quiet** parameter that suppresses all output to screen.

A switch variable is generally preferred over a **Boolean** variable for parameters because the syntax for users is simpler. The users do not need to include a **\$true** or **\$false** value.

Default values

You can define default values for parameters in the **Param()** block. The default values that you define are only used if the user does not provide a value for the parameter. This ensures that every necessary parameter has a value.

The example below shows how to set a default value:

```
Param(
    [string]$ComputerName = "LON-DC1"
)
```

Requesting user input

You can also prompt for input if the user does not provide a parameter value. This ensures that the user provides a value for a parameter when there is not a logical default value that you can specify.

The example below shows how to prompt users for input:

```
Param(
    [int]$EventID = Read-Host "Enter event ID"
)
```



Note: To enable more advanced functionality for parameters, such as specifying a parameter as mandatory, there is a **Parameter()** attribute that you can use. The advanced functionality enabled by the **Parameter()** attribute is covered in course 10962C, *Advanced Automated Administration with Windows PowerShell*.

Demonstration: Obtaining user input by using parameters

In this demonstration, you will see how to obtain user input by using parameters.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. Rename **E:\Mod09\Democode\10961C_Mod09_Demo02.txt** to **10961C_Mod09_Demo02.ps1**.
3. Open Windows PowerShell Integrated Scripting Environment (ISE) and open **E:\Mod09\Democode\10961C_Mod09_Demo02.ps1**.
4. Review the code.
5. To set the current directory, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod09\Democode
```

6. To pass values to the script by position, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo02.ps1 LON-DC1 300
```

7. To pass values to the script by parameter name, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo02.ps1 -EventID 300 -ComputerName LON-DC1
```

8. To view the results when no parameter data is provided, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo02.ps1
```

9. In Windows PowerShell ISE, modify line 2 to ask for user input when a computer name is not supplied.

10. Modify line 3 to set a default value of **300** when an event ID is not specified.

11. Save the script.

12. To view the results when no parameter data is provided, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo02.ps1
```

13. When prompted for a computer name, type **LON-DC1**, and then press Enter.

14. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Question: Why is it useful to assign default values to parameters in a script?

Lesson 2

Overview of script documentation

Adding comments and help documentation to your script make it easier for others to use. Comments are used to explain what is happening in different parts of a script, to aid in maintenance and modifications. Help documentation makes it easier to use a script by providing help information in the same format that is available for cmdlets when you use **Get-Help**. In this lesson, you learn about comments and adding help information to a script.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to document a script by using comments.
- Explain how to add comments to a script.
- Explain how to add help information.
- Adding help information to a script.

Using comments to document a script

Comments are text that you can put in scripts to provide additional information. Using comments is not a technical requirement, but comment make scripts easier to understand. For example, you can include comments that describe:

- What a variable is used for.
- What task a loop is performing.
- Why a task was performed a specific way.
- The end of a set of braces for a loop.

Your initial goal for including comments should be to provide enough information for someone else to understand how the script works. Even if the script is only for your own use, you should include comments in this format. Having comments in the script will save you time if you need to modify the script in the future. Many administrators have written a script and then needed to modify it six months or a year later, and then must expend significant effort to figure out what they did in the script. By using comments, that process is much faster.

You can also use scripts as part of troubleshooting. Placing some of your script code in comments stops that code from being executed and can help you isolate the part of the script that is generating an error.

- Comments make scripts easier to understand and can be used to describe:
 - What a variable is used for
 - What task a loop is performing
 - A task was performed a specific way
 - The end of a set of braces for a loop
- Comments can be used to remove a section of code during troubleshooting
- Comments can be:
 - Single line: #
 - Blocks: <# #>

Single-line comments

A single-line comment is identified by the pound symbol (#). If you place a pound symbol at the start of a line, Windows PowerShell does not attempt to process the line. If you place a pound symbol partway through a line, anything past the pound symbol is not processed.

The following example shows a single-line comment:

```
# This section disables the selected user accounts
```

Studijní materiály Okškolení

Block comments

If you have a large section of code that you want to comment out, or a large amount of text you want to include as a comment, a block comment might be easier to use than multiple single-line comments. A block comment begins with `<#` and ends with `#>`. Everything between the start and end points of the block comment is not processed.

The following example shows a block comment:

```
<# All of this text  
Is part of a single  
Block comment #>
```

Demonstration: Adding comments to a script

In this demonstration, you will see how to add comments to a script.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. Rename `E:\Mod09\Democode\10961C_Mod09_Demo03.txt` to `10961C_Mod09_Demo03.ps1`.
3. Open Windows PowerShell ISE and open `E:\Mod09\Democode\10961C_Mod09_Demo03.ps1`.
4. Review the code.
5. At the Windows PowerShell prompt, change the current directory to `E:\Mod09\Democode`.
6. To view the script output, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo03.ps1
```

7. In Windows PowerShell ISE, add the comment **Information for troubleshooting** on line 7 and save the script.
8. To verify that the script output has not changed, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo03.ps1
```

9. In Windows PowerShell ISE, configure a block quote around lines 9 and 10 and save the script.
10. To verify that troubleshooting information is no longer displayed, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo03.ps1
```

11. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Adding help information

To make your script functionality more discoverable to users, you can configure help information in the script. Users view the help information by using the **Get-Help** cmdlet just as they do with Windows PowerShell cmdlets. This consistency makes the system easy for users.

Help information is placed at the top of a script in a block comment. The block comment ensures that Windows PowerShell does not try to process the help information, but keeps it discoverable by the **Get-Help** cmdlet.

Within the block comment, keywords define the sections of help information. Each section starts with a period (.) and the keyword that describes the section. The following table lists the valid keywords in the expected order.

```

Adding comment-based help makes script functionality
more discoverable for users:
<#
.SYNOPSIS
Disables user accounts that have not signed in for a specified
number of days.
.DESCRIPTION
Use this script to disable users accounts that have not signed
in for an extended period of time. Review the output to
ensure that no unintended accounts were disabled.
.PARAMETER Days
The number of days for which users have not signed in.
.EXAMPLE
.\DisableStaleUsers.ps1 -Days 60
#>

```

Keyword	Description
SYNOPSIS	A brief description of what the script does.
DESCRIPTION	A detailed description of what the script does. The PARAMETER keyword can be used within the description to describe each parameter. There should be one instance of the PARAMETER keyword for each parameter that is described.
EXAMPLE	Examples and descriptions of how to use the script. There can be multiple examples, and each example uses one instance of the EXAMPLE keyword.
INPUTS	Describes the type of objects that the script accepts from the pipeline.
OUTPUTS	Describes the type of objects that the script returns.
NOTES	A section for additional information about the script that is not contained elsewhere.
LINK	Can contain the name of a related topic or a URL to a web-based copy of the help file. If there are multiple related topics, each has a LINK keyword.
COMPONENT	Describes a technology for a feature that the script uses.
ROLE	Describes the user role for the help topic.
FUNCTIONALITY	Describes the intended use of the function. There are additional keywords that can be used in this section to further describe the script.

The keywords are not case sensitive and can be placed in any order. The order of the keywords in the table above is used in the help information by convention and is not a technical requirement.

Some help is displayed only when specific parameters are used with **Get-Help**. For example, **Get-Help -Component** displays the help identified by the COMPONENT keyword.

The syntax for adding comment-based help is below:

```
<#
.SYNOPSIS
Disables user accounts that have not signed in for a specified number of days.
.DESCRIPTION
Use this script to disable users accounts that have not signed in for an extended period
of time. Review the output to ensure that no unintended accounts were disabled.
.PARAMETER Days
The number of days for which users have not signed in.
.EXAMPLE
.\DisableStaleUsers.ps1 -Days 60
#>
```

Demonstration: Adding help information to a script

In this demonstration, you will see how to add help information to a script.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. Rename **E:\Mod09\Democode\10961C_Mod09_Demo04.txt** to **Query-Bios.ps1**.
3. Open Windows PowerShell ISE.
4. In Windows PowerShell ISE, open **Query-Bios.ps1**.
5. In Windows PowerShell ISE, open **E:\Mod09\Democode\10961C_Mod09_Demo04_Help.txt**.
6. Copy the text from **10961C_Mod09_Demo04_Help.txt** and paste it at the top of **Query-Bios.ps1**.
7. At the Windows PowerShell prompt, change the current directory to **E:\Mod09\Democode**.
8. To view basic help information, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Help .\Query-Bios.ps1
```

9. To see the examples in the help, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Help .\Query-Bios.ps1 -Examples
```

10. To view all the help information, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Help .\Query-Bios.ps1 -Full
```

11. Close the Windows PowerShell ISE window and the Windows PowerShell prompt.

Question: Why is it important to add comments within scripts?

Lab A: Accepting data from users

Scenario

You often find that you need to review the disk space available on your organization's servers. Rather than connecting to each server and viewing the available disk space, you are writing a script that queries remote disk utilization.

Objectives

After completing this lab, you should be able to:

- Query disk information by using a script.
- Configure a script to use alternate credentials.
- Document a script.

Lab Setup

Estimated Time: **60 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available VM environment. Before beginning the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the VM starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Exercise 1: Querying disk information from remote computers

Scenario

Your script for disk information has the following requirements:

- Accept the remote computer name as a parameter.
- If no computer name is provided as a parameter, the user should be prompted to enter a computer name.
- The query for information should use Web Services-Management (WS-MAN) rather than Distributed Component Object Model (DCOM).
- Show logical disk information (volumes) rather than physical disk information.
- Only information for local disks (hard drives) should be included.

Studijní materiály Okškolení

The main task for this exercise is as follows:

1. Create a script that queries disk information with current credentials.

► **Task 1: Create a script that queries disk information with current credentials**

1. Perform all tasks using **LON-CL1**.
2. Identify the cmdlet that allows you to remotely query hardware information by using WS-MAN.
3. Identify the syntax required to query logical disk information and only include local disks.
4. Create a new script **E:\Mod09\QueryDisk.ps1**.
5. Add a **param()** block to the script that accepts a computer name and prompts for a computer name if one is not provided.
6. Verify that the script correctly queries disk information from **LON-DC1**.

Results: After completing this exercise, you will have created a script to query disk information from remote computers.

Exercise 2: Updating the script to use alternate credentials

Scenario

You would like to run a script that queries disk information from remote computers. To account for scenarios where the user does not have permission to query disk information on remote server, you are updating the script to accept alternate credentials when specified.

The script needs to meet the following requirements:

- Accept a switch parameter to indicate whether alternate credentials are required.
- If alternate credentials are required, gather, and use those credentials.

The main task for this exercise is as follows:

1. Update the script to use alternate credentials.

► **Task 1: Update the script to use alternate credentials**

1. Update the **param()** block to include a switch that indicates alternate credentials will be used.
2. Add an **if** statement that evaluates the switch.
 - If the switch is true, run new code that gathers alternate credentials
 - If the switch is false, run the existing query.
3. For new code, you must:
 - Get the credential from the user.
 - Open a remote session using that credential.
 - Send the query using that session.

Results: After completing this exercise, you will have updated the script to accept alternate credentials.

Exercise 3: Documenting a script

Scenario

After completing your script, you want to document it to make it easier to use in the future.

The main tasks for this exercise are as follows:

1. Document a script.
2. Prepare for the next lab.

► Task 1: Document a script

1. Review the code in your script, and add comments that make it easier to understand.
2. Add comment-based help to your script. At a minimum, it should include the following sections:
 - o SYNOPSIS
 - o DESCRIPTION
 - o PARAMETER
 - o EXAMPLE
3. Use **Get-Help** to verify that you can query the help information.

► Task 2: Prepare for the next lab

- Keep the virtual machines running as you will need them for the next lab.

Results: After completing this exercise, you will have documented a script.

Question: You configured a switch parameter to indicate whether alternate credentials are required. What are the possible values for a switch parameter?

Question: Why did you use **Get-CimInstance** instead of **Get-WmiObject** in this script?

Lesson 3

Troubleshooting and error handling

As you develop more scripts, you will find that efficient troubleshooting makes your script development much faster. A big part of efficient troubleshooting is understanding error messages. For some difficult problems, you can use breakpoints to stop a script partway through execution to query variables values. To make scripts more user friendly, you can use **Try..Catch** to handle errors during script execution.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how error messages are stored.
- Explain how to add additional troubleshooting information to scripts.
- Describe how to configure breakpoints for troubleshooting.
- Explain how to troubleshoot a script.
- Describe error actions in Windows PowerShell.
- Describe how to use **Try..Catch** for error handling.
- Explain how to identify specific errors for use with **Try..Catch**.
- Use **Try..Catch** to handle errors.

Understanding error messages

When you run Windows PowerShell commands, you will sometimes encounter error messages.

Large red error messages on the screen can be intimidating at first, but they contain information that you can use for troubleshooting. Most of the time, if you read the complete message, it provides a good description of why the error occurred. Errors can occur for reasons such as:

- You made a mistake typing.
- You queried an object that does not exist.
- You attempted to communicate with a computer that is offline.

- Error messages are useful for troubleshooting
- Some causes of error messages:
 - You made a mistake typing
 - You queried an object that does not exist
 - You attempted to communicate with a computer that is offline
- Errors are stored in the **\$Error** array
 - The most recent error is stored in **\$Error[0]**

When errors occur, they are stored in an **\$Error** array. The most recent error is always at index zero. When a new error is generated, it is inserted at **\$Error[0]**, and the index of other errors is increased by one. It can be useful to review errors in **\$Error** whenever you need to look back at a previous error message. For example, if you clear the screen, you can use **\$error** to look at the most recent error message.

Adding script output

When a script is not operating as you expected, it can be useful to have the script display additional information. You can use that information to understand what the script is doing and why it is not operating as expected.

The **Write-Host** cmdlet is the most common way to display additional information while a script is running. You can use **Write-Host** to display text information that indicates specific points in a script and variable values. Variable values can be useful in most cases when a script is not behaving as you expect it to behave, because a variable does not have a value that you expect.

- Use **Write-Host** to display additional information when the script is executing:
 - Variable values
 - Location in the script
- To slow down the data onscreen for easier viewing, use:
 - **Start-Sleep**
 - **Read-Host**
- Comment out the **Write-Host** commands when not required

If you want to slow down execution of a script to enable you to read the output better, you can add a **Start-Sleep** cmdlet and specify a few seconds to pause. Alternatively, if you want the script to pause until you are ready for it to continue, you can use **Read-Host**.

While you are in the process of troubleshooting, you can comment out the additional information. Then, if required, you can uncomment it to review the additional information again.



Note: If you have configured your script as an advanced script by using **CmdletBinding()**, you can also use **Write-Verbose** and **Write-Debug** as part of your script for troubleshooting. The advanced functionality enabled by **CmdletBinding()** is covered in course 10962C, *Advanced Automated Administration with Windows PowerShell*.

Using breakpoints

A breakpoint pauses a script and provides an interactive prompt. At the interactive prompt, you can query or modify variable values and then continue the script. You use breakpoints to troubleshoot scripts when they are not behaving as expected.

At a Windows PowerShell prompt, you can set breakpoints by using the **Set-PSBreakPoint** cmdlet. Breakpoints can be set based on the line of the script, a specific command being used, or a specific variable being used. The example below shows how to set a breakpoint at a specific line of a script:

```
Set-PSBreakPoint -Script "MyScript.ps1" -Line 23
```

- Use breakpoints to pause a script and query or modify variable values
- **Set-PSBreakPoint** examples:
 - Set-PSBreakPoint -Line 23 -Script "MyScript.ps1"
 - Set-PSBreakPoint -Command "Set-ADUser" - Script "MyScript.ps1"
 - Set-PSBreakPoint -Variable "computer" -Mode ReadWrite -Script "MyScript.ps1"
- You use the **-Action** parameter to specify code to perform
- The Windows PowerShell ISE has line-based breakpoints

Studyjní materiály Okskolení

When you set a breakpoint based on a line, you need to be careful when you edit the script. As you edit a script, you might add or remove lines, and the breakpoint will not affect the same code that you initially intended. The following example shows how to set a breakpoint for a specific command:

```
Set-PSBreakPoint -Command "Set-ADUser" -Script "MyScript.ps1"
```

When you set a breakpoint based on a command, you can include wildcards. For example, you could use the value **"*-ADUser"** to trigger a breakpoint for **Get-ADUser**, **Set-ADUser**, **New-ADUser**, and **Remove-ADUser**. To set a breakpoint for a specific variable, do the following:

```
Set-PSBreakPoint -Variable "computer" -Script "MyScript.ps1" -Mode ReadWrite
```

You can use the *-Mode* parameter for variables to identify whether you want to break when the variable value is read, written, or both. Valid values are **Read**, **Write**, and **ReadWrite**.

The default action for **Set-PSBreakPoint** is break, which provides the interactive prompt. However, you can use the *-Action* parameter to specify code that runs instead. This allows you to perform complex operations such as evaluating variable values and only breaking if the value is outside a specific range.

 **Note:** Breakpoints are stored in memory rather than as part of the script. Breakpoints are not shared between multiple Windows PowerShell prompts and are removed when prompt is closed.

Windows PowerShell ISE

In the Windows PowerShell ISE, you can set breakpoints based on line. Options related to breakpoints are in the Debug menu. Lines that you configure as breakpoints are highlighted, making it easy to identify them. Also, in the Windows PowerShell ISE, as you add or remove lines to your script, the breakpoints are updated automatically with the correct line number.

Demonstration: Troubleshooting a script

In this demonstration, you will see how to troubleshoot a script.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. Rename **E:\Mod09\Democode\10961C_Mod09_Demo05.txt** to **10961C_Mod09_Demo05.ps1**.
3. Open Windows PowerShell ISE and open **10961C_Mod09_Demo05.ps1**.
4. In Windows PowerShell ISE, review the code.
5. To set the current directory, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod09\Democode
```

6. To view the script output, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo05.ps1
```

7. To view the error, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$Error[0]
```

8. To clear the **\$Error** variable, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$Error.Clear()
```

9. To create a breakpoint, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-PSBreakPoint .\10961C_Mod09_Demo05.ps1 -Line 5
```

10. To run the script, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo05.ps1
```

11. To view the value **\$ComputerName**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$ComputerName
```

12. To test the value **\$ComputerName**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$ComputerName -eq $null
```

13. To test the value **\$ComputerName**, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$ComputerName -eq ""
```

14. To exit the debug prompt, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
exit
```

15. In Windows PowerShell ISE, on line 5, change **\$null** to "", and then save the file.

16. To view all breakpoints, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-PSBreakPoint
```

17. To remove all breakpoints, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-PSBreakPoint | Remove-PSBreakPoint
```

18. To run the script, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo05.ps1
```

19. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Understanding error actions

When a PowerShell command generates an error, that error might be one of two types, either a *terminating* error or a *non-terminating* error. A terminating error occurs when Windows PowerShell determines that it is not possible to continue processing after the error and the command stops. A non-terminating error occurs when Windows PowerShell determines that it is possible to continue processing after the error. When a non-terminating error occurs, script execution or pipeline execution will continue. Non-terminating errors are more common than terminating errors.

Consider the following command:

```
Get-WmiObject -Class Win32_BIOS -ComputerName LON-SVR1,LON-DC1
```

If you assume that the computer **LON-SVR1** is not available on the network, **Get-WmiObject** will generate an error when it tries to query that computer. However, the command could continue with the next computer, **LON-DC1**. The error is therefore a non-terminating error.

\$ErrorActionPreference

Windows PowerShell has a built-in, global variable named **\$ErrorActionPreference**. When a command generates a non-terminating error, the command checks that variable to see what it should do. The variable can have one of the following four possible values:

- **Continue** is the default, and it tells the command to display an error message and continue to run.
- **SilentlyContinue** tells the command to display no error message, but to continue running.
- **Inquire** tells the command to display a prompt asking the user what to do.
- **Stop** tells the command to treat the error as terminating and to stop running.

To set the **\$ErrorActionPreference** variable:

```
$ErrorActionPreference = 'Inquire'
```



Note: Be selective about using **SilentlyContinue** for **\$ErrorActionPreference**. You might think that this makes your script better for users, but it could make troubleshooting difficult.

If you intend to trap an error within your script so that you can handle the error, commands must use the **Stop** action. You can trap and handle only terminating errors. However, it is considered a poor practice to change **\$ErrorActionPreference** if the error that you expect will be produced by a Windows PowerShell command.

-ErrorAction parameter

All Windows PowerShell commands have the *-ErrorAction* parameter. This parameter has the alias *-EA*. The parameter accepts the same values as **\$ErrorActionPreference**, and the parameter overrides the variable for that command. If you expect an error to occur on a command, you therefore use *-ErrorAction* to set that command's error action to **Stop**. Doing this lets you trap and handle the error for that command, but leaves all other commands to use the action in **\$ErrorActionPreference**. An example follows:

```
Get-WmiObject -Class Win32_BIOS -ComputerName LON-SVR1,LON-DC1 -ErrorAction Stop
```

The only time that you usually will modify **\$ErrorActionPreference** is when you expect an error outside of a Windows PowerShell command, such as when you are executing a method such as the following:

```
Get-Process -Name Notepad | ForEach-Object { $PSItem.Kill() }
```

In this example, the **Kill()** method might generate an error. But because it is not a Windows PowerShell command, it does not have the *-ErrorAction* parameter. You would instead set **\$ErrorActionPreference** to **Stop** before running the method, and then set the variable back to **Continue** after you run the method.

Using Try..Catch

If your script generates a non-terminating error, by default it will display a large red error message and continue processing. For users of your script the large red error message can be alarming.

Users might think that your script is poor quality based on receiving that error message.

You can implement error handling in your scripts to improve the user experience. Error handling can be used to provide user friendly error messages or log errors to a file. To implement error handling, you use **Try..Catch**. The following example shows how error handling can be implemented:

```
Try {
    Get-WmiObject -Class Win32_BIOS -ComputerName $comp -ErrorAction Stop
} Catch {
    Write-Host "Error connecting to $comp"
} Finally {
    Write-Host "BIOS query for $comp is complete"
}
```

- Use **Try..Catch** to implement error handling to:

- Create user-friendly error messages
- Write errors to a log file

```
Try {
    Get-WmiObject -Class Win32_BIOS -
        ComputerName $comp -ErrorAction Stop
} Catch {
    Write-Host "Error connecting to $comp"
} Finally {
    Write-Host "BIOS query for $comp is complete"
}
```

The following table describes the three code blocks from the example above.

Block	Description
Try	This block contains the code that is monitored for an error. If a terminating error occurs while running this code, the Catch block runs. In the example above, the error action is set to Stop to force non-terminating errors to become terminating errors. The error message that would normally be written to screen is suppressed.
Catch	This block runs only if a terminating error is generated in the Try section. This is the block where you can place a more user-friendly error or write the error to a log file. In the example above, this block is displaying a user-friendly error message when the Try code generates an error.
Finally	This block runs whether there is an error or not. The Finally block is optional and is seldom used. If the example above is used in a ForEach loop to process an array of computers then the status information provided by the Finally block would be displayed on screen for all computers regardless of whether there was an error.



Note: Some cmdlets, such as **Get-WmiObject**, can accept an array of objects as input. However, when using **Try..Catch**, you should process only one item at a time to clearly identify when an error occurs. Instead of passing an array to a cmdlet, use a **ForEach** loop to process each item in the array individually.

When displaying output after catching an error, you can generate your own customized error message by using **Write-Error**. This cmdlet displays customizable output similar to a typical Windows PowerShell error using red text.

Identifying specific errors to use with Try..Catch

Simple scripts generally have simple errors, and one **Catch** block is sufficient for error handling. However, as scripting gets more sophisticated, you might want to perform different actions for different errors related to the same event. For example, when you script attempts to create a log file, there might be multiple potential errors, such as the directory does not exist, the file already exists, or you do not have the correct permissions to create a file in that location. Instead of a generic error indicating that the log file could not be created, you could have an error message specific to each scenario. Or, you could write code that remediates the error.

- You can implement **Catch** blocks for specific error types

```
Try{
    New-Item $file
} Catch [System.IO.DirectoryNotFoundException] {
    Write-Host "Directory was not found"
} Catch [System.IO.IOException] {
    Write-Host "The file already exists"
} Catch {
    Write-Host "An unknown error occurred"
}
```

- Find the specific error type by using:
\$Error[0].Exception.GetType().FullName

Studijní materiály Oskolení

You can implement multiple **Catch** blocks for a single **Try** block. To identify when each **Catch** block should be used, you need to define one or more error types for each of the **Catch** blocks. The following example shows multiple **Catch** blocks being used:

```
Try {
    New-Item $file
} Catch [System.IO.DirectoryNotFoundException] {
    Write-Host "Directory was not found"
} Catch [System.IO.IOException] {
    Write-Host "The file already exists"
} Catch {
    Write-Host "An unknown error occurred"
}
```

The error type required for a **Catch** block can be obtained from the **\$Error** array. After you simulate the error, use the following to view the error type:

```
$Error[0].Exception.GetType().FullName
```

 **Note:** The error generated by the **Try** block is available in the **Catch** block as **\$_** or **\$PSItem**.

Demonstration: Handling errors

In this demonstration, you will see how to handle errors.

Demonstration Steps

1. Click **Start**, type **power**, and click **Windows PowerShell**.
2. Rename **E:\Mod09\Democode\10961C_Mod09_Demo06.txt** to **10961C_Mod09_Demo06.ps1**.
3. Click **Start**, type **ise**, and click **Windows PowerShell ISE**.
4. In Windows PowerShell ISE, click the **File** menu and click **Open**.
5. In the **Open** window, in the address bar, type **E:\Mod09\Democode** and press Enter.
6. Click **10961C_Mod09_Demo06.ps1** and click **Open**.
7. Review the code in the script and note that Section 1 with no error checking is the current code. Section 2 with error checking has a block comment around it.
8. Run the script and notice that an error is generated because **LON-SVR1** is not available.
9. Create a block comment around Section 1.
10. Remove the block comment around Section 2.
11. Run the script and notice that the error is replaced by a message generated by error handling.
12. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Question: What is the purpose of using breakpoints?

Lesson 4

Functions and modules

When you create many scripts, you will have snippets of code that you want to reuse. You will also have snippets of code that you want to reuse within the same script. Rather than having the same code appear multiple times in a script, you can create a function that is called multiple times. If you need to use the same code across multiple scripts, then you can put the function into a module that can be shared by multiple scripts. In this lesson, you will learn to create functions and modules.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe functions.
- Describe the implications of variable scope.
- Explain how to use dot sourcing.
- Create a function in a script.
- Explain how to create a module.
- Create a module.

What are functions?

A function is a block of reusable code. You can use functions to perform repetitive actions within a script rather than putting the same code in the script multiple times. For example, if you have a large script that can perform multiple actions, rather than putting code that logs data to disk with each action, you can have one function that logs data to disk. Then the logging function is called each time an action is performed. Later if you want to change that logging function it only needs to be changed in one place.

When you call a function, you can pass data to it.

You use the a **Param()** block for a function in the same way as you do for a script. After the declaration for the function, insert the **Param()** block and the definitions for any variables that are expected to be passed to the function. The following example is a function that uses a **Param()** block to accept a computer name:

```
Function Get-SecurityEvent {
    Param (
        [string]$ComputerName
    ) #end Param
    Get-EventLog -LogName Security -ComputerName -$ComputerName -Newest 10
}
```

To call the function within a script, use the following syntax:

```
Get-SecurityEvent -ComputerName LON-DC1
```

- Use functions to perform repetitive tasks in scripts
- Pass data to a function by using a **Param()** block:


```
Function Get-SecurityEvent {
    Param (
        [string]$ComputerName
    ) #end Param
    Get-EventLog -LogName Security
    -ComputerName -$ComputerName -Newest 10
}
```
- To call a function:


```
Get-SecurityEvent -ComputerName LON-DC1
```
- To store the results of a function:


```
$SecurityEvents=Get-SecurityEvent
-ComputerName LON-DC1
```

In the example above, the value for `-Computer` parameter is passed to the `$Computer` variable in the function. **Get-EventLog** then queries the most recent 10 events from the security log of that computer and displays them on the screen. If you want those events placed in a variable and available for use in the remainder of the script, use this syntax:

```
$securityEvents = Get-SecurityEvent -ComputerName LON-DC1
```

Using variable scopes

Intuitively, you would assume that the variable named `$computer` that you set in a function could be accessed in the script when the function is complete. However, that is not the case. Variables have a specific scope and are limited in how they interact between scopes.

The following table describes the three scopes and how they affect variable use.

- There are three scopes for variables:
 - Global
 - Script
 - Function
- Avoid using the same variable name in multiple scopes
- You can modify a variable in a higher scope by specifically referencing the scope:
`$script:var="Modified from function"`
- Set a script scope variable equal to the function output instead
 - Use `Return()` to pass a variable value back

Scope	Description
Global	<p>The global scope is for the Windows PowerShell prompt. Variables set at the Windows PowerShell prompt can be read in all the scripts started at that Windows PowerShell prompt.</p> <p>Variables created at a Windows PowerShell prompt do not exist in other Windows PowerShell prompts or in instances of the Windows PowerShell ISE.</p>
Script	<p>The script scope is for a single script. Variables set within a script can be read by all the functions within that script.</p> <p>If you set a variable value in the script scope that already exists in the global scope, a new variable is created in the script scope. There are then two variables of the same name in two separate scopes. At this point, when you view the value of the variable in the script, the value of the variable in the script scope is returned.</p>
Function	<p>The function scope is for a single function. Variables set within a function are not shared with other functions or the script.</p> <p>If you set a variable value in the function scope that already exists in the global or script scope, a new variable is created in the function scope. There are then two variables of the same name in two separate scopes.</p>

 **Note:** To avoid confusion, it is a best practice to avoid using the same variable names in different scopes.

In addition to reading a variable in a higher-level scope, you can also modify that variable by specifically referencing the scope of the variable when you modify it. To modify a script scope variable from a function, use the following syntax:

```
$script:var = "Modified from function"
```

It is a best practice to avoid modifying variables between scopes because doing so can cause confusion. Instead, set the script scope variable equal to the output of the function. If the data in the function is in a variable, you can use **Return()** to pass it back to the script.

Following is an example of using **Return()** at the end of a function to pass a variable value back to the script scope:

```
Return($users)
```

 **Note:** Using **Return()** in a function adds the specified data to the pipeline of data being returned, but does not replace existing data in the pipeline. As part of script development, you need to verify exactly which data is being returned by a function.

Demonstration: Creating a function in a script

In this demonstration, you will see how to create a function.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. Rename **E:\Mod09\Democode\10961C_Mod09_Demo07.txt** to **10961C_Mod09_Demo07.ps1**.
3. Click **Start**, type **ise**, and click **Windows PowerShell ISE**.
4. Open Windows PowerShell ISE and open **10961C_Mod09_Demo07.ps1**.
5. Review the code.
6. To set the prompt location, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod09\Democode
```

7. To view the size of a folder, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
.\10961C_Mod09_Demo07.ps1 -Path "C:\Windows"
```

8. To view the size of a folder including subfolders, at the Windows PowerShell prompt, type the following command, and then press Enter:
9. In Windows PowerShell ISE, to create a function, add **Function Get-FolderSize {** as the first line and **}** as the last line.
10. To call the function, at the end of the file add **Get-FolderSize -Path "C:\Program Files" -Recurse**.

11. To call the function a second time, at the end of the file, add **Get-FolderSize -Path C:\Windows**.
12. Save the script, and then run it to review the results.
13. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Creating a module

You can create modules for storing functions.

After you put your functions into modules, they are discoverable just as cmdlets are. Also, like the modules included with Windows, the modules you create load automatically when a function is required.

 **Note:** As a best practice, you should name your functions in modules with a naming structure similar to cmdlet naming convention. For example, you would use the verb-noun format.

- Functions should use standard cmdlet naming conventions
- Modules use the **.psm1** file extension
- Module locations are stored in **\$PSModulePath** environmental variable:
 - C:\Users\ UserID\Document\WindowsPowerShell\Modules
 - C:\Program Files\WindowsPowerShell\Modules
 - C:\Windows\system32\WindowsPowerShell\1.0\Modules
- Module files are placed in a subfolder of the same name:
 - C:\Program Files\WindowsPowerShell\Modules\AdatumFunctions\AdatumFunctions.psm1

 **Note:** Functions in modules can include comment-based help that is discoverable by using **Get-Help**. The help information is included in each function.

In many cases, you already have your functions in a Windows PowerShell script file. To convert that script file to a module, rename it with the **.psm1** file extension. No structural changes in the file are required.

Windows PowerShell uses the **\$PSModulePath** environmental variable to define the paths from which modules are loaded. In Windows PowerShell 5.0, the following paths are listed:

- **C:\Users\ UserID\Document\WindowsPowerShell\Modules**
- **C:\Program Files\WindowsPowerShell\Modules**
- **C:\Windows\System32\WindowsPowerShell\1.0\Modules**

 **Note:** If you store modules in **C:\Users\ UserID\Document\WindowsPowerShell\Modules**, they are only available to a single user.

Modules are not placed directly in the Modules directory. Instead, you must create a subfolder with the same name as the file and place the file in that folder. For example, if you have a module named **AdatumFunctions.psm1**, you would place it in **C:\Program Files\WindowsPowerShell\Modules\AdatumFunctions**.

Studyjní materiály pro skolení

Demonstration: Creating a module from a function

In this demonstration, you will see how to create a module.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. To set the prompt location, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod09\Democode\
```

3. To copy and rename a script file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Copy-Item .\10961C_Mod09_Demo07.ps1 .\FolderFunctions.psm1
```

4. Open Windows PowerShell ISE and open **FolderFunctions.psm1**.
5. Review the code.
6. Remove the last two lines that call the function and save the file.
7. To create a folder for the module, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
New-Item -Type Directory "C:\Program Files\WindowsPowerShell\Modules\FolderFunctions"
```

8. To copy the .psm1 file, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Copy-Item .\FolderFunctions.psm1 "C:\Program  
Files\WindowsPowerShell\Modules\FolderFunctions"
```

9. To verify that the module is recognized, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Module -ListAvailable F*
```

10. To verify that the module is not loaded, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Module
```

11. To use the function in the module, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-FolderSize -Path C:\Windows
```

12. To verify that the module is loaded, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Module
```

13. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Studijní materiály Okskolení

Using dot sourcing

Dot sourcing is a method for importing another script into the current scope. If you have a script file that contains functions, you can use dot sourcing to load the functions into memory at a Windows PowerShell prompt. Normally, when you run the script file with functions, the functions are removed from memory when the script completes. When you use dot sourcing, the functions remain in memory and you can use them at the Windows PowerShell prompt. You can also use dot sourcing within a script to import content from another script.

Dot sourcing can load from a local file or over the network by using a Universal Naming Convention (UNC) path. The syntax for using dot sourcing is shown below:

```
. C:\scripts\functions.ps1
```

At one time, dot sourcing was the only method available to maintain a centralized repository of functions that could be reused across multiple scripts. However, modules are a more standardized and preferred method for maintaining functions used across scripts.

Question: Why would you prefer to use script modules for functions instead of dot sourcing?

Lab B: Implementing functions and modules

Scenario

In this lab, you are creating a logging function that you can use in scripts to record errors or other information.

Objectives

After completing this lab, you will be able to:

- Create a function.
- Convert a function to a module.

Lab Setup

Estimated Time: **60 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available VM environment. Before beginning the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the VM starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Exercise 1: Creating a logging function

Scenario

The logging function that you are creating has the following requirements:

- Accepts a parameter for the file name for the log file.
- Accepts a parameter for the folder containing the log file.
- Accepts a parameter for the data being written to the log file.
- Adds a timestamp to the data being written to the log file.
- Appends each log entry to an existing file.

The main task for this exercise is as follows:

- Create a logging function.

► **Task 1: Create a logging function**

1. Create a new script file named **LogFunction.ps1**.
2. In the script, create a function named **Write-Log**.
3. In the **Write-Log** function, create a **Param()** block that accepts the folder, file name, and data.
4. Calculate the full path for the log file and place it in a variable.
5. Calculate a timestamp that includes a short version of the date and a long version of the time.
6. Send the timestamp and data to the file.
7. Add a line to the script that calls the function, and then run the script to test it.

Results: After completing this exercise, you should have created a logging function.

Exercise 2: Adding error handling to a script

Scenario

After using the function for a period, you realize that two common errors occur when using the function. First, the folder for the log file needs to have a trailing backslash. Second, the folder might not exist. To address each of these issues:

- Review the folder path provided and add a backslash if necessary.
- If an error is generated when writing the data to disk, generate a friendly error message.

The main task for this exercise is as follows:

1. Add error handling to a script.

► **Task 1: Add error handling to a script**

1. At the beginning of the function, after the **Param()** block, add a test to identify whether the folder provided ends with a backslash (\).
2. If the test is false, add a backslash to the folder value.
3. Verify that the function works when a backslash is not provided in the folder path.
4. Use a **Try..Catch** construct to provide a friendly error message if writing to file fails.
5. Verify that a friendly error message appears when an invalid path is used.

Results: After completing this exercise, you will have added error handling to a script.

Exercise 3: Converting a function to a module

Scenario

To simplify using the **Write-Log** function in multiple scripts, you are converting it to a function that any script can access.

The main tasks for this exercise are as follows:

1. Convert a function to a module.
2. Prepare for the next module.

► Task 1: Convert a function to a module

1. In the script, remove the line that calls the function. Only the function should be in the file.
2. Save the script as **C:\Program Files\WindowsPowerShell\Modules\LogFunction\LogFunction.psm1**.
3. At a Windows PowerShell prompt, confirm that **Write-Log** is available for use.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Results: After completing this exercise, you will have converted a function to a module.

Question: If you use **Try..Catch** to create a friendly error message, what type of information should be included in that error message?

Question: How do you call a function from within a script?

Module Review and Takeaways

Review Questions

Question: Is it possible to store credentials to disk for later reuse?

Question: Is it possible to use `Try..Catch` and provide different responses for different errors?

Studijní materiály Okškolení

Studijní materiály Okškolení

Module 10

Administering remote computers

Contents:

Module Overview	10-1
Lesson 1: Using basic Windows PowerShell remoting	10-3
Lesson 2: Using advanced Windows PowerShell remoting techniques	10-13
Lab A: Using basic remoting	10-19
Lesson 3: Using PSSessions	10-22
Lab B: Using PSSessions	10-28
Module Review and Takeaways	10-31

Module Overview

Windows PowerShell remoting is a technology that enables you to connect to one or more remote computers and instruct them to run commands on your behalf. It has become the foundation for administrative communications in a Windows-based operating system environment. Windows PowerShell 5.0 is installed as part of the Windows Management Framework 5.0, which offers both Windows PowerShell functionality and remoting capabilities. For example, the graphical Server Manager console in Windows Server 2016 relies on remoting to communicate with servers—even to communicate with the local computer on which the console is running. Past versions of the operating system have included remoting, but its use was largely optional. Now, the technology is quickly becoming a mandatory component of every environment.

The earlier modules discussed how the Windows Management Framework 5 is included with Windows Server 2016, and it updates Windows PowerShell Desired State Configuration (Windows PowerShell DSC), Windows Remote Management (WinRM), and Windows Management Instrumentation (WMI).

In this module, you will use remoting mainly in its default configuration, using HTTP on port 5985. You can configure remoting to allow for or to require encryption based on Secure Sockets Layer (SSL), by using the HTTPS protocol, which in this case defaults to port 5986 rather than the traditional port 443. However, to use HTTPS, a receiving computer must be configured to have an SSL certificate, which makes the remoting configuration more complex.



Additional Reading: For remoting documentation, refer to: "About Remote" at:
<https://aka.ms/jbml20>

Objectives

After completing this module, you will be able to:

- Describe remoting architecture and security.
- Use advanced Windows PowerShell remoting techniques.
- Create and manage persistent PSSessions.

Lesson 1

Using basic Windows PowerShell remoting

Although remoting is a complex technology, working with Windows PowerShell remoting is fairly straightforward once you understand the underlying concepts. In this lesson, you learn how to use remoting to perform administration on remote computers.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Windows PowerShell remoting architecture.
- Explain the difference between Windows PowerShell remoting and other forms of remote administration.
- Describe Windows PowerShell remoting security and privacy features.
- Enable remoting on a computer.
- Use Windows PowerShell remoting for single computer management.
- Use Windows PowerShell remoting for multiple computer management.
- Use Windows PowerShell remoting.
- Explain the difference between local output and remoting output.

Remoting overview and architecture

Remoting uses an open standard protocol called Web Services for Management (WS-Management or WS-MAN). As the name implies, this protocol is built on the same HTTP (or HTTPS) protocol that web browsers use to communicate with web servers. This makes the protocol easier to control and to route through firewalls. The Windows operating system implements the protocol in the WinRM service.

You must enable remoting on the computers that you want to receive incoming connections, although no configuration is necessary on computers that are initiating outgoing connections. The technology is enabled by default for incoming connections on Windows Server 2012 (and later server operating systems) and can be enabled on any computer that is running Windows PowerShell 2.0 or later. Additionally, Windows PowerShell 2.0 and later can communicate with one another in a mixed-version environment.

Remoting:

- Uses WS-MAN protocol, using HTTP (by default) or HTTPS
- Is implemented by WinRM service
- Is enabled by default on Windows Server 2016; available on any computer running Windows PowerShell 2.0 or greater
- Is not enabled on any client operating system
- Must be enabled on any computer that will receive incoming connections



Note: Although remoting is enabled by default on Windows Server 2012 and later Windows Server operating systems, it is not enabled by default on client operating systems, including Windows 8 and Windows 10.

Studijní materiály Okškolení

Windows PowerShell remoting uses WinRM, which can manage communications for other applications. For example, on a default Windows Server 2016 installation, WinRM handles communications for 64-bit Windows PowerShell, 32-bit Windows PowerShell, and two Server Manager components.

Architecture

Remoting starts with the WinRM service. It registers one or more listeners. Each listener accepts incoming traffic through either HTTP or HTTPS, and each listener can be bound to a single local IP address or to multiple IP addresses. There is no dependency on Microsoft Internet Information Services (IIS). This means that IIS does not have to be installed for WinRM to function.

Incoming traffic includes a packet header that indicates the traffic's intended destination, or *endpoint*. In Windows PowerShell, these endpoints are also known as *session configurations*. Each endpoint is associated with a specific application, and when traffic is directed to an endpoint, WinRM starts the associated application, hands off the incoming traffic, and waits for the application to complete its task. The application can pass data back to WinRM, and WinRM transmits the data back to the originating computer.

In a Windows PowerShell scenario, you would send commands to WinRM, which then executes the commands. (The process is listed as Wsmprovhost in the remote computer's process list.) Windows PowerShell would then execute those commands, and convert (or *serialize*) the resulting objects—if there are any—into XML. The XML text stream is then handed back to WinRM, which transmits it to the originating computer. Windows PowerShell on the remote computer deserializes the XML back into static objects. This enables the command results to behave much like any other objects within the Windows PowerShell pipeline.

Windows PowerShell can register multiple endpoints, or session configurations, with WinRM. In fact, a 64-bit operating system will register an endpoint for both the 64-bit Windows PowerShell host and the 32-bit host. This is by default. You also can create your own custom endpoints that have highly precise permissions and capabilities assigned to them.

Remoting vs. remote connectivity

Remoting is the name of a specific Windows PowerShell feature. Do not confuse it with the more generic concept of *remote connectivity*.

Many commands implement their own communications protocols, although in the future many of them may be changed to use remoting instead. For example, **Get-WmiObject** uses Remote Procedure Calls (RPCs), whereas **Get-Process** communicates with the computer's Remote Registry Service. Microsoft Exchange Server commands have their own communications channels, and Active Directory commands talk to a specific web service gateway by using their own protocol. All these other forms of communication may have unique firewall requirements and may require specific configurations to be in place to operate.

Remoting is a generalized way to transmit *any* command to a remote computer for local execution. The command that you execute does not have to be present on the computer that initiates the connection; only the remote computers must be able to know about the command and to run it.

- *Remoting* is the name of a specific feature that utilizes a specific service and protocol
- When used in Windows PowerShell, use the term *Windows PowerShell remoting*
- It applies to a relatively small subset of commands that can communicate with remote computers
- A command with a **-ComputerName** parameter does not necessarily mean it uses remoting
- Nonremoting commands use their own protocols:
 - RPCs, which include WMI
 - Remote Registry Service (for example, **Get-Process**)

The purpose of remoting is to reduce or eliminate the need for individual command authors to code their own communications protocols. Many command authors are already required to do this to ship their products. This is why many different protocols and technologies are being used currently.

We recommend you use the entire term *Windows PowerShell remoting* in a Windows PowerShell environment.

Remoting security

By default, the endpoints that Windows PowerShell created only allow connections by members of a particular group. On Windows Server 2016 and Windows 10, there are two groups: the Remote Management Users group, and the local Administrators group. Because domain administrators are members of every domain computers' local Administrators group, this applies to those accounts as well. On earlier operating system versions, members of the local Administrators group are allowed by default. In common practice, this means that only domain administrators can use remoting, unless you add other groups or users to the Remote Management Users group. Each endpoint does have a System Access Control List (SACL) that you can change to control exactly who can connect to it.

- Remoting is security transparent; you can perform only those tasks that your credentials allow
- Mutual authentication helps prevent delegation of credentials to spoofed or impersonated computers:
 - It works in domain environments by default
 - It can use SSL in lieu of domain credentials
 - It can be disabled through the TrustedHosts list

The default remoting behavior is to delegate your sign-in credentials to the remote computer, although you do have the option of specifying alternative credentials when you make a connection. Regardless, the remote computer uses those credentials to impersonate you and perform the tasks that you have specified by using those credentials. If you have enabled auditing, in addition to the tasks that you perform, the tasks that remoting performs on your behalf will also be audited. In effect, remoting is security transparent, it does not change your environment's existing security. With remoting, you can perform all the tasks that you would perform while physically located in front of the local computer.

Security risks and mutual authentication

Delegating your credential to a remote computer involves some security risks. For example, if an attacker successfully impersonates a known remote computer, you could potentially hand over highly privileged credentials to that attacker, who could then use them for malicious purposes. Because of this risk, remoting by default requires *mutual authentication*, which means that you must not only authenticate yourself to the remote computer, the remote computer must also authenticate itself to you. This guarantees that you connect only to the exact computer that you intended.

Mutual authentication is a native feature of the Active Directory Kerberos authentication protocol, and when you connect between trusted domain computers mutual authentication occurs automatically. When you connect to nondomain computers, you have to either provide another form of mutual authentication in the form of an SSL certificate (and the HTTPS protocol that must be set up in advance), or turn off the requirement for mutual authentication by adding the remote computer to your local TrustedHosts list.

Note however, that TrustedHost uses Windows NT LAN Manager (NTLM) authentication. NTLM authentication does not ensure server identity. As with any protocol using NTLM for authentication, attackers who have access to a domain-joined computer's trusted account could cause the domain controller to create an NTLM session-key and thus impersonate the server.

Studyjní materiály pro skolení

Computer name considerations

Because remoting must be able to look up a remote computer in Active Directory Domain Services (AD DS), you must refer to computers only by their canonical computer name. IP addresses or DNS aliases, for example, will not work, because they do not provide remoting with the mutual authentication that remoting needs. If you must refer to a computer by IP address or by a DNS alias, either you must connect by using HTTPS (meaning the remote computer must be configured to accept that protocol), or you must add the IP address or DNS alias to your local TrustedHosts list.

 **Note:** A special exception is made for the computer name **localhost**, which enables you to use it to connect to the local computer without any other configuration changes. If the local computer is using a client-based operating system, then WinRM needs to be configured on it.

The TrustedHosts list

The *TrustedHosts* list is a locally configured setting that you also can configure by using a Group Policy Object (GPO). The TrustedHosts list lists the computers for which mutual authentication is not possible. Computers must be listed with the same name that you will use to connect to them, whether that be an actual computer name, a DNS alias, or an IP address. You can use wildcards to specify SRV*, which allows any computer whose name or DNS alias starts with "SRV" to connect. However, use caution with this list. Although it does make connecting to nondomain computers (without having to set up HTTPS) easier, the TrustedHosts list bypasses an important security measure. It allows you to send your credential—potentially a highly privileged one—to a remote computer without determining whether that computer is in fact one that you intended to connect to. You should use the TrustedHosts list only to designate computers that you know for a fact cannot be easily impersonated or compromised, such as servers housed in a protected datacenter. You also can use TrustedHosts to temporarily enable connections to nondomain computers on a controlled network subnet, such as new computers that are undergoing a provisioning process.

 **Best Practice:** Avoid using the TrustedHosts list unless absolutely necessary. Configuring a nondomain computer to use HTTPS is a more secure long-term solution.

Privacy

By default, remoting uses HTTP, which does not offer privacy (encryption) for the contents of your communications. However, Windows PowerShell can and does apply application-level encryption by default. This means that your communications receive a degree of privacy and protection. On internal networks, this application-level encryption is generally sufficient for most organizations' requirements.

In a domain environment that uses the default Kerberos authentication protocol, credentials are sent in the form of encrypted Kerberos tickets that do not include passwords.

When you connect by using HTTPS, the entire channel is encrypted by using the encryption keys of the remote computer's SSL certificate so that even if you use Basic authentication, passwords do not transmit in clear text. However, when you connect by using HTTP and the Basic authentication protocol, to a computer that is not configured for HTTPS, credentials might be transmitted in clear text, including passwords. This can occur when you connect to a nondomain computer that you add to your local TrustedHosts list, or even when you use a domain-joined computer by specifying its IP address rather than its host name.

Because credentials must be passed in clear text in that scenario, you should ensure that you connect to a nondomain computer only on a controlled and protected network subnet, such as a subnet specifically designated for new computer provisioning. If you have to routinely connect to a nondomain computer, you should configure it to support HTTPS or it will default to NTLM authentication.

Enabling remoting

Be aware that you need to enable Windows PowerShell remoting only on computers that will receive incoming connections. No configuration is necessary to enable outgoing communications (except to make sure that any local firewall will allow the outgoing traffic).

Manually enabling remoting

To manually enable Windows PowerShell remoting on a computer, run the Windows PowerShell **Enable-PSRemoting** cmdlet. A member of the local Administrators group must perform this task. This is a persistent change (you can disable it later by running **Disable-PSRemoting**). This command will:

- Create an exception in the Windows Firewall for incoming TCP traffic on port 5985.
- Create an HTTP listener on port 5985 for all local IP addresses.
- Set the WinRM service to start automatically and then restart if necessary.
- Register default endpoints for use by Windows PowerShell.

- To enable Windows PowerShell remoting manually, run **Enable-PSRemoting** as an Administrator
- To enable Windows PowerShell remoting centrally, configure a GPO
- There are restrictions on client computers where a network connection is set to Public
- Windows Server 2012 and later enable Windows PowerShell remoting by default; no further steps are needed

This command will fail on client computers where one or more network connections are set to Public (instead of Work or Home). You can override this failure by adding the **-SkipNetworkProfileCheck** parameter. However, be aware that the Windows Firewall will not allow exceptions when you are connected to a Public network.

Enabling remoting by using a GPO

Many organizations will prefer to centrally control Windows PowerShell remoting enablement and settings through GPOs. Microsoft does support this. You must set up various settings in a GPO to duplicate the steps taken by **Enable-PSRemoting**. The procedure is not too difficult; however, it does involve several steps, and thus is not under the purview of this course.

Using remoting: one-to-one

One-to-one remoting resembles the Secure Shell (SSH) tool that is used on many UNIX and Linux computers, in that you get a command prompt on the remote computer. The exact operation of remoting is quite different from SSH. However, the end effect and usage is almost the same. In Windows PowerShell, you type commands on your local computer, which then transmits them to the remote computer for execution. Results are serialized into XML and transmitted back to your computer, which then deserializes them into objects and puts them into the Windows PowerShell pipeline. Unlike SSH, one-to-one remoting is not built on the Telnet protocol.

One-to-one Windows PowerShell remoting is similar in concept to SSH, although different in actual operation:

1. Start with **Enter-PSSession -ComputerName name**
2. The Windows PowerShell prompt changes to indicate the connected computer
3. Exit with **Exit-PSSession**

You enable one-to-one Windows PowerShell remoting by using the **Enter-PSSession** command, combined with its **-ComputerName** parameter. You can use other parameters to perform basic customization of the connection; these are covered later in this module.

After you are connected, the Windows PowerShell prompt changes to indicate the computer to which you are connected. Run **Exit-PSSession** to exit the session and return to the local command prompt. If you close Windows PowerShell while connected, the connection will close on its own.

Using remoting: one-to-many

The process of converting an object into a form that can be readily transported is known as **Serialization**. Serialization takes an object's state and transforms it into serial data format, such as XML or binary format. Deserialization converts the formatted XML or binary data into an object type.

One-to-many remoting lets you send a single command to multiple computers in parallel. Each computer will execute the command that you transmit, serialize the results into XML, and transmit those results back to your computer.

Your computer deserializes the XML into objects, and then puts them into the Windows PowerShell pipeline. When doing this, several properties are added to each object. This includes a **PSComputerName** property that indicates which computer each result came from. That property lets you sort, group, and filter based on computer name.

- **Invoke-Command** can send a command or script to one or more remote computers in parallel
- Results include a **PSComputerName** property that indicate the computer each result came from
- Considerations include:
 - Throttling
 - Passing data to remote computers
 - Persistence
 - Ways to specify computer names

Usage scenarios

You can use one-to-many remoting in two main ways:

- **Invoke-Command -ComputerName name1,name2 -ScriptBlock { command }**

This technique sends either the command or the commands contained in the script block to the computers that are listed. This technique is useful for sending one or two commands (multiple commands are separated by a semicolon).

- **Invoke-Command -ComputerName *name1,name2* -FilePath *filepath***

This technique sends the contents of a script file (with a .ps1 file name extension) to the computers that are listed. The local computer opens the file and reads its contents. However, the remote computers do not have to have direct access to the file. This technique is useful for sending a large file of commands, such as a complete script.



Note: Within any script block, including the script block provided to the **-ScriptBlock** parameter, you can use a semicolon (;) to separate multiple commands. For example, **{ Get-Service ; Get-Process }** will run **Get-Service**, and then run **Get-Process**.

Throttling

By default, Windows PowerShell will connect to only 32 computers at once. If you list more than 32 computers, the excess computers will be queued. As some initial computers complete and return their results, computers will be pulled from the queue and contacted.

You can alter this behavior by using the **-ThrottleLimit** parameter of **Invoke-Command**. Raising the number does not put additional load on the remote computers. However, it does put an additional load on the computer where **Invoke-Command** was run and uses more bandwidth. Each concurrent connection is basically a thread of Windows PowerShell. Therefore, raising the number of computers consumes memory and processor on the local computer.

Passing values

The contents of the script block or file are transmitted as literal text to the remote computers that run them exactly as is. The computer does not parse the script block or file on which the **Invoke-Command** was run. Consider the following example:

```
$var = 'BITS'
Invoke-Command -ScriptBlock { Get-Service -Name $var } -Computer LON-DC1
```

In this scenario, the variable **\$var** is being set on the local computer, named **SERVER2**. The value of **\$var** is not inserted into the script block. In other words, this command asks **LON-DC1** to retrieve a service whose name is equal to the one in the variable **\$var**. **SERVER2**, however, does not know the value of **\$var**, because the value has not been defined on LON-DC1. This is a common mistake that administrators new to Windows PowerShell often make. There is a specific way to manage this situation, and it will be covered in the next lesson.

Local execution and remote execution

Pay close attention to the commands that you enclose in the script block, which will be passed to the remote computer. Remember that your local computer will not do any processing with the contents of the script block. Those contents will be passed as is to the remote computer. For example, consider this command:

```
Invoke-Command -ScriptBlock { Do-Something -Credential (Get-Credential) }
-ComputerName LON-DC1
```

This command will run the **Get-Credential** cmdlet on the remote computer. If you try running **Get-Credential** on a local computer it will use a graphical dialog box to prompt for the credential. Will that work when run on a remote computer? If you ran the preceding command on 100 remote computers, would you be prompted for 100 credentials?

Now consider this modified version of the command:

```
Invoke-Command -ScriptBlock { Param($c) Do-Something -Credential $c }
    -ComputerName LON-DC1
    -ArgumentList (Get-Credential)
```

This command runs **Get-Credential** on the local computer, and runs it only once. The resulting object is passed into the **\$c** parameter of the script block, enabling each computer to use the same credential.

These examples illustrate the importance of writing remoting commands carefully. By using a combination of remote execution and local execution, you can achieve a variety of useful goals.



Note: You will learn more about credential objects in Module 12, "Using Profiles and Advanced Windows PowerShell Techniques."

Persistence

Using the techniques outlined here, every time you use **Invoke-Command** the remote computer creates a new wsmprovhost process, runs the command or commands, returns the results, and then closes that Windows PowerShell instance. Each successive **Invoke-Command**, even if made to the same computer, is like opening a new Windows PowerShell window. Any work done by a previous session will not exist unless you save it to a disk or some other persistent storage. For example:

```
Invoke-Command -ComputerName LON-DC1 -ScriptBlock { $x = 'BITS' }
Invoke-Command -ComputerName LON-DC1 -ScriptBlock { Get-Service -Name $x }
```

In this example, the second **Invoke-Command** would fail, because it is dependant on a variable that was created in a previous wsmprovhost process. When the first **Invoke-Command** finished running, that variable was lost. You can create a wsmprovhost process on a remote computer so that you can successfully send successive commands to it, and you will learn about that technique later in this module.

Multiple computer names

The **-ComputerName** parameter of **Invoke-Command** can accept any collection of string objects as computer names. The following list describes different techniques:

- **-ComputerName ONE,TWO,THREE**

This is a static, comma-separated list of computer names.

- **-ComputerName (Get-Content Names.txt)**

This reads names from a text file that is named Names.txt, assuming the file contains one computer name per line.

- **-ComputerName (Import-Csv Computers.csv | Select -ExpandProperty Computer)**

This reads a comma-separated values (CSV) file that is named Computers.csv. It contains a column named **Computer** that contains computer names.

- **-ComputerName (Get-ADComputer -Filter * | Select -ExpandProperty Name)**

This queries every computer object in Active Directory Domain Services (AD DS) (which can take significant time in a large domain).

Common mistakes when using computer names

Be careful where you specify a computer name. For example:

```
Invoke-Command -ScriptBlock { Get-Service -ComputerName ONE,TWO }
```

This command does not provide a **-ComputerName** parameter to **Invoke-Command**. Therefore, the command runs on the local computer. The local computer will run **Get-Service**, and tell **Get-Service** to connect to computers named **ONE** and **TWO**. The protocols used by **Get-Service** will be used; Windows PowerShell remoting will not be used. Compare that with the following command:

```
Invoke-Command -ScriptBlock { Get-Service } -ComputerName ONE,TWO
```

This command will use Windows PowerShell remoting to connect to computers named **ONE** and **TWO**. Each of these computers will run **Get-Service** locally, returning their results by means of remoting.

Additional Windows PowerShell remoting

For more interactive Windows PowerShell remoting situations, you can manage sessions as their own entity by first creating a session by using the **New-PSSession** command. The benefit to using the **New-PSSession** command is the session will persist throughout multiple **Invoke-Command** instances, allowing you to pass variables and objects to other commands in your script. You can create persistent sessions by using the **New-PSSession** command and assigning it to a variable, which you can later reference using the **Invoke-Command** command, and when finished, then close using the **Remove-PSSession** command. You will learn more about this command in Lesson 3 of this module.

Demonstration: Enabling and using remoting

In this demonstration, you will see how to enable remoting on a client computer, and how to use remoting in several basic scenarios.

Demonstration Steps

1. Ensure that you have the correct execution policy in place by running the command **Set-ExecutionPolicy RemoteSigned**.
2. Enable remoting.
3. Open a one-to-one connection to **LON-DC1**.
4. Get a list of processes running on **LON-DC1**.
5. Close the **LON-DC1** connection.
6. Get a list of the most recent 10 Security event log entries from both **LON-CL1** and **LON-DC1**.

Remoting output vs. local output

When you run a command such as **Get-Process** on your local computer, the command puts objects—in this case, objects of the type **System.Diagnostics.Process**—in the Windows PowerShell pipeline. These objects have properties, methods, and frequently events. Methods make the object do some task. The **Kill()** method of a **Process** object, for example, ends the process that the object represents.

When a command runs on a remote computer, that computer serializes the results in XML, and transmits that XML text to your computer. You do this to put the object's information into a format that can be transmitted over a network. However, for complex objects, the serialization procedure can deal only with static information about an object—in other words, its properties.

When the XML is received by your computer, the XML is deserialized back into objects that are put in the Windows PowerShell pipeline. When you have a **Process** object, by piping it to **Get-Member**, you know it is now of the type **Deserialized.System.Diagnostics.Process**, a related, but different, kind of object. The deserialized object has no methods and no events.

Hence, from a practical perspective, you should consider any data that is received through remoting to be a static snapshot. The data is not updatable, and the objects cannot be used to take any actions. Therefore, you should do as much processing as possible on the remote computer, where the objects are still live objects that have methods and events.

For example, the following example is a bad command:

```
Invoke-Command -Computer LON-DC1 -ScriptBlock { Get-Process -Name Note* } | Stop-Process
```

In this example, you are receiving back **Process** objects; the action of stopping will occur on the local computer and not the remote computer. This action stops any local processes that happened to have names matching the remote ones.

The correct approach would be as follows:

```
Invoke-Command -Computer LON-DC1 -ScriptBlock { Get-Process -Name Note* | Stop-Process }
```

In this approach, the processing has occurred entirely on the remote computer, with only the final results being serialized and sent back. The difference between these two commands is subtle but important to understand.

Question: Why would an administrator decide to use remoting instead of managing a computer directly?

Question: What are some security concerns with remoting?

- Results received through remoting are deserialized from XML
- As a result, they are not live objects and do not have methods or events
- As a strategy, try to have as much processing as possible occur on the remote computer, with only the final results coming back to you through remoting

Lesson 2

Using advanced Windows PowerShell remoting techniques

Windows PowerShell remoting includes several advanced techniques that help achieve specific goals, or alleviate specific shortcomings. In the previous lesson, you reviewed the shortcomings that some of the basic techniques generate. In this lesson, you will learn about some useful advanced techniques that will help overcome these challenges.

Lesson Objectives

After completing this lesson, you will be able to:

- Configure common remoting options.
- Send parameters and local variables to remote computers.
- Describe the use of the **\$Using:** modifier.
- Send local variables to a remote computer.
- Configure multi-hop remoting authentication.

Common remoting options

Both **Enter-PSSession** and **Invoke-Command** support several parameters that can let you change common connection options. These include the following:

- **–Port** specifies an alternative TCP port for the connection. Use this when the computer to which you are connecting is listening on a port other than the default 5985 (HTTP) or 5986 (HTTPS). Be aware that you can, locally or through Group Policy, configure a different port as a permanent new default.
- **–UseSSL** instructs Windows PowerShell to use HTTPS instead of HTTP.
- **–Credential** specifies an alternative credential for the connection. This credential will be validated by the remote computer, and must have sufficient privileges and permissions to perform whatever tasks you intend to perform on the remote computer.
- **–ConfigurationName** connects to an endpoint (session configuration) other than the default endpoint. For example, you can specify **microsoft.powershell32** to connect to the remote computer's 32-bit Windows PowerShell endpoint.
- **–Authentication** specifies an authentication protocol. The default is Kerberos, and other options include Basic, CredSSP, Digest, Negotiate, and NegotiateWithImplicitCredential. The protocol that you specify must be enabled in the WS-MAN configuration on both the initiating and receiving computers.

- **–Port**
- **–UseSSL**
- **–Credential**
- **–ConfigurationName**
- **–Authentication**

Additional options are available by creating a **PSSessionOption** object, and then passing it to **–SessionOption**

Studijní materiály Okškolení

You can configure additional session options by using **New-PSSessionOption** to create a new session option object, and then passing it to the **-SessionOption** parameter of **Enter-PSSession** or **Invoke-Command**. Review the Help file for **New-PSSessionOption** to learn about its capabilities.

You can modify the default values, such as the port number and enabled authentication protocols in the WSMAN PSDrive.

Sending parameters to remote computers

You have already learned that **Invoke-Command** cannot include variables in its script block or script file, unless the remote computer can understand those variables. Therefore, it might seem more complicated to find a way to pass data from the initiating computer to the remote computer. However, **Invoke-Command** actually provides a specific mechanism for doing this task.

To review, the intent behind the following command is to display a list of the 10 most recent Security event log entries on each targeted computer. However, the command will not work as written:

```
$Log = 'Security'
$Quantity = 10
Invoke-Command -Computer ONE,TWO -ScriptBlock {
    Get-EventLog -LogName $Log -Newest $Quantity
}
```

- You cannot use local variables in the **Invoke-Command** script block
- You can pass data. However, you must use a specific technique
- Pass local variables to the **-ArgumentList** parameter of **Invoke-Command**; they will map to variables in a **Param()** block inside the script block

The problem is that the variables **\$Log** and **\$Quantity** have meanings only on the local computer, and those values are not inserted into the script block prior to those values being sent to the remote computers. The remote computers do not know what they mean.

The correct syntax for this command is as follows:

```
$Log = 'Security'
$Quantity = 10
Invoke-Command -Computer ONE,TWO -ScriptBlock {
    Param($x,$y) Get-EventLog -LogName $x -Newest $y
} -ArgumentList $Log,$Quantity
```

By using this command, the local variables are passed to the **ArgumentList** parameter of **Invoke-Command**. Within the script block, a **Param()** block is created, which contains the same number of variables as the **-ArgumentList** list of values—that is, two. You can name the variables within the **Param()** block with any names that you want. They will receive data from the **ArgumentList** parameter based on order. In other words, because **\$Log** was listed first on **ArgumentList**, its value will be passed to **\$x** because it is first in the **Param()** block. The variables in the **Param()** block can then be used inside the script block, as shown in the example.



Note: The syntax shown in these examples will work for Windows PowerShell 2.0 and later. However, Windows PowerShell 3.0 introduced a simplified alternative syntax. If you have a local variable **\$variable**, and you want to include its contents in a command that will be executed on a remote computer, you can run the following command:

```
Invoke-Command -ScriptBlock { Do-Something $Using:variable } -ComputerName REMOTE
```

The special **\$Using:** prefix is understood by the local computer, and **\$Using:variable** will be replaced with the contents of the local variable **\$variable**.

This same technique works with the **-FilePath** parameter of **Invoke-Command**. In that case, Windows PowerShell expects the script file to already contain a **Param()** block and will attach the **ArgumentList** values in the order in which they are listed.

Windows PowerShell scopes

Windows PowerShell provides access protection to variables, aliases, functions, and Windows PowerShell drives by limiting where they can be changed and read. By enforcing a few simple rules using Windows PowerShell scopes, you ensure that you do not inadvertently change an item that should not be changed.

The basic rules of scopes are:

- Unless you explicitly make it private, items you include in a scope are visible in the scope in which it was created and in any child scope. You can use variables, aliases, functions, or Windows PowerShell drives in one or more scopes.
- Items you create within a scope can be changed only in the scope in which the item was created, unless you explicitly stipulate a different scope.

- Scopes provides access protection to variables, aliases, functions, and Windows PowerShell drives:
 - Limits where they can be changed and read
 - Ensures you do not inadvertently change an item
- The scope modifier identifies a local variable in a remote command
- The syntax of this modifier is **\$Using:**

```
$ps = "Windows PowerShell"
Invoke-Command -ComputerName LON-DC1 -ScriptBlock {Get-WinEvent -LogName $Using:ps}
```

If you create an item in a scope, and the item has the same name with an item in a different scope, the original item might get hidden under the new item, but does not overridden or change the original item.

You can use local variables in remote commands, but you must indicate that the variable is defined in the local session. Windows PowerShell assumes that the variables used in remote commands are defined in the session in which the command runs.

The **\$Using:** scope modifier

Beginning with Windows PowerShell 3.0, you can implement the **\$Using:** scope modifier to identify a local variable in a remote command. This is a special scope modifier and is the preferred and easiest way to accomplish getting a local variable to a remote command. This technique passes on the variable value(s) to the remote computer, and therefore invokes less processing across the hosts being used. By default, variables in remote commands are assumed to be defined in the remote session.

The syntax of Using is: **\$Using:**

Studijni materiály Okškolení

In the following example, the `$ps` variable is created in the local session, but is used in the session in which the command runs. The `$Using:` scope modifier identifies `$ps` as a local variable:

```
$ps = "Windows PowerShell"
Invoke-Command -ComputerName LON-DC1 -ScriptBlock {Get-WinEvent -LogName $Using:ps}
```

You can also apply the `$Using:` scope modifier in PSSessions such as below:

```
$s = New-PSSession -ComputerName LON-DC1
$ps = "Windows PowerShell"
Invoke-Command -Sessions $s -ScriptBlock {Get-WinEvent -LogName $Using:ps}
```

Demonstration: Sending local variables to a remote computer

In this demonstration, you will see two ways to pass local information to a remote computer by using **Invoke-Command**, first with the **-ArgumentList** parameter and then with the `$Using:` scope modifier.

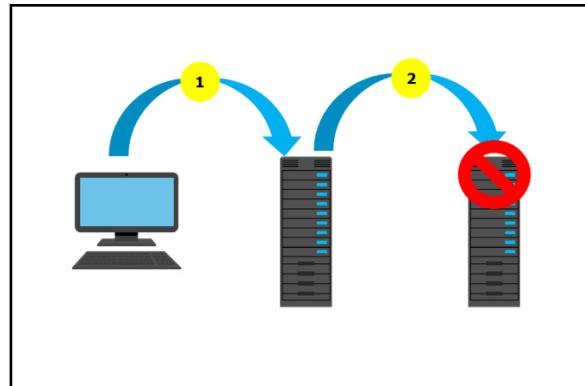
Demonstration Steps

1. Create local variables to hold data.
2. Pass the local data to the remote computers to customize command execution.
3. First use the **-ArgumentList** parameter and then use the `$Using:` scope modifier.

Multi-hop remoting

One issue with remoting is how remote computers delegate credentials. By default, credentials can be delegated across only one connection, or *hop*. This single delegation prevents the remote computer from further delegating your credentials, which could present a security risk.

However, the need to perform multiple hop, or *multi-hop*, delegation can often occur in production environments. For example, in some organizations administrators are not permitted to connect directly from their client computers to a server in the data center. Instead, they must connect to an intermediate gateway or jump server, and then from there connect to the server they intend to manage. In its default configuration, remoting does not permit this approach. After you are connected to a remote computer, your credential can go no further than the remote computer. Trying to access any resource that is not located on that computer typically results in a failure because your access is not accompanied by a credential. The solution is to enable Credential Security Support Provider (CredSSP).



Enabling CredSSP

You must enable the CredSSP protocol both on the initiating computer, referred to as *the client*, and on the receiving computer, referred to as *the server*. Doing this enables the receiving computer to delegate your credential one additional hop. However, there have been numerous security breaches documented while using CredSSP, and it is no longer a preferred option. If the server you authenticate to use CredSSP is compromised, the credentials you pass through CredSSP are also compromised. You should instead use constrained delegation.

To configure the client, run the following command, substituting *servername* with the name of the server that will be able to redelegate your credential:

```
Enable-WsManCredSSP -Role Client -Delegate servername
```

The server name can contain wildcard characters, although using only the asterisk (*) wildcard is too permissive, because you would be enabling any computer to redelegate your credential, even an unauthorized user. Instead, consider a limited wildcard pattern, such as *.ADATUM.com, that would limit redelegation to computers in that domain.

To configure the server, run **Enable-WsManCredSSP –Role Server**. No delegated computer list is needed on the server.

You also can configure these settings through Group Policy, which offers a more centralized and consistent configuration across an enterprise.

Resource-based Kerberos constrained delegation

When using Windows Server 2012 or newer server operating systems, you can forgo using CredSSP and instead use constrained delegation. Constrained delegation, included in Windows Server 2012 and newer, introduces controlling delegation of service tickets by using security descriptors rather than an allow list of service principal names (SPN). This change simplifies delegation by allowing the resource to determine which security principals can request tickets on behalf of another user. Resource-based constrained delegation works correctly regardless of domain functional level and the number of domain controllers running a version of Windows Server prior to Windows Server 2012. Constrained delegation needs:

- At least one Windows Server 2012 or higher domain controller in the same domain as the host computer from which the Windows PowerShell remoting commands are being run.
- One Windows Server 2012 or higher domain controller in the domain hosting the remote server you are trying to access from the intermediate remote server.

The code for setting up the permissions requires Windows Server 2012 or higher computer with the Windows Server 2012 Active Directory PowerShell Remote Server Administration Tools (RSAT).

You can add the RSAT as a Windows feature by running the following two commands:

```
Add-WindowsFeature RSAT-AD-PowerShell  
Import-Module ActiveDirectory
```

To grant resource-based Kerberos constrained delegation from **LON-SVR1** through **LON-SVR2** to **LON-SVR3**, run the following command:

```
Set-ADComputer -Identity LON-SVR2 -PrincipalsAllowedToDelegateToAccount LON-SVR3
```

One issue could cause this command to fail. The Key Distribution Center (KDC) has a 15 minute SPN negative cache. If **LON-SVR2** has already tried to communicate with **LON-SVR3**, then there is a negative cache entry. You will need to clear the cache on **LON-SVR2** by using one of the following techniques:

- Run the command **klist purge -li 0x3e7** (preferred and fastest method).
- Wait 15 minutes for the cache to clear automatically.
- Reboot **LON-SVR2**.

To test constrained delegation, run the following code example:

```
# Capture a credential
$cred = Get-Credential Adatum\TestUser
# Test Kerberos double hop
Invoke-Command -ComputerName LON-SVR1.Name -Credential $cred -ScriptBlock {
    Test-Path \\$($using:ServerC.Name)\C$
    Get-Process lsass -ComputerName $($using:LON-SVR2.Name)
    Get-EventLog -LogName System -Newest 3 -ComputerName $($using:LON-SVR3.Name)
}
```

Question: Why might you configure remoting to use ports other than the default ports?

Lab A: Using basic remoting

Scenario

You are an administrator for Adatum Corporation and must perform some maintenance tasks on a server. You do not have physical access to the server, and instead plan to perform the maintenance tasks by using Windows PowerShell remoting. The server runs Windows Server 2016. You also have some tasks that must be performed against both a server and another client computer that runs Windows 10.

Objectives

After completing this lab, you will be able to:

- Enable remoting on a client computer.
- Execute a task on a remote computer by using one-to-one remoting.
- Execute a task on two computers by using one-to-many remoting.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must follow these steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Microsoft Hyper-V Manager, click **10961C-LON-DC1**, and in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

Perform the lab steps on the **10961C-LON-CL1** virtual machine.

Exercise 1: Enabling remoting on the local computer

Scenario

In this exercise, you will enable remoting on the client computer.

The main task for this exercise is to enable remoting for incoming connections.

► Task: Enable remoting for incoming connections

1. Ensure that you are signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd** with the Windows PowerShell command window open.
2. Ensure the **ExecutionPolicy** is set to **RemoteSigned**.

Studijní materiály Okškolení

3. Enable remoting for incoming connections on the **LON-CL1** client computer.
4. Verify that Windows PowerShell has created several session configurations. You have to find a command that can produce this list.

Results: After completing this exercise, you should have enabled remoting on the client computer.

Exercise 2: Performing one-to-one remoting

Scenario

In this exercise, you will connect to a remote computer and perform maintenance tasks.

The main tasks for this exercise are as follows:

1. Connect to the remote computer and install an operating system feature on it.
2. Test multi-hop remoting.
3. Observe remoting limitations.

► Task 1: Connect to the remote computer and install an operating system feature on it

1. Ensure that you are still signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. On **LON-CL1**, in the Windows PowerShell console, use remoting to establish a one-to-one connection to **LON-DC1**.
3. Install the Network Load Balancing (NLB) feature on **LON-DC1**.
4. Disconnect from **LON-DC1**.

► Task 2: Test multi-hop remoting

1. Establish a one-to-one remoting connection with **LON-DC1**.
2. Try to establish a connection from **LON-DC1** to **LON-CL1**. What happens and why?
3. Close the connection.

► Task 3: Observe remoting limitations

1. Ensure that you are signed in to the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. Establish a one-to-one remoting connection to **LON-CL1** using the computer name **localhost**. This is your local computer, but this new connection creates a second user session for you on the computer.
3. Use Windows PowerShell to start a new instance of Notepad. What happens? Why?
4. Stop the pipeline to exit the Notepad process.
5. Close the connection.

Results: After completing this exercise, you should have connected to a remote computer, and performed maintenance tasks on it.

Exercise 3: Performing one-to-many remoting

Scenario

In this exercise, you will run commands against multiple computers. One of those will be the client computer, although you will be establishing a second sign-in to it for the duration of each command.

The main tasks for this exercise are as follows:

1. Retrieve a list of physical network adapters from two computers.
 2. Compare the output of a local command to that of a remote command.
 3. Prepare for the next lab
- **Task 1: Retrieve a list of physical network adapters from two computers**
1. Ensure that you are still signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
 2. On **LON-CL1**, using a keyword such as **adapter**, find a command that can list network adapters.
 3. Read the Help for the command, then find a switch parameter that will limit output to physical adapters.
 4. Use remoting to run the command on **LON-CL1** and **LON-DC1**.
- **Task 2: Compare the output of a local command to that of a remote command**
1. Pipe a collection of **Process** objects to **Get-Member**.
 2. Use remoting to retrieve a list of **Process** objects from **LON-DC1**, and then pipe them to **Get-Member**.
 3. Compare the output of the two **Get-Member** results.

Results: After completing this exercise, you will have run commands against multiple remote computers.

► **Task 3: Prepare for the next lab**

- Keep the virtual machines running for the next lab. Do not revert the virtual machines.

Question: You established a PSSession from **LON-CL1** to **LON-DC1**, and then within that PSSession, you tried to establish a PSSession back to **LON-CL1**. This failed. Why?

Lesson 3

Using PSSessions

To this point in this module, you learnt that each remoting command that you used created a connection, used it, and then closed it. However, as previously discussed this format does not offer any kind of persistence of information on the remote computer. Using a persistent connection enables you to interactively query and manage a remote computer. In this lesson, you will learn how to establish and manage persistent connections to remote computers, known as Windows PowerShell sessions or PSSessions.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of persistent connections.
- Create a PSSession.
- Explain how to use a PSSession.
- Transmit commands by using a PSSession.
- Explain how to disconnect from PSSessions.
- Disconnect and reconnect to PSSessions.
- Explain the concept of implicit remoting.
- Execute remote commands by using implicit remoting.

Persistent connections

To this point, when you have run **Enter-PSSession** or **Invoke-Command**, Windows PowerShell has had to make the connection to the remote computer, run the commands that you specified, then return the results to Windows PowerShell, and then close the connection. This technique offers no persistence of information across connections, because each connection is essentially starting all over again.

Windows PowerShell does have the capability of creating persistent connections, which are known as *sessions*, or more accurately, *PSSessions*. (The PS designation signifies Windows PowerShell and differentiates these sessions from other kinds of sessions that might be present in other technologies, such as a Remote Desktop Services session.)

After making a PSSession to a remote computer, you run the desired commands within the session, but then you leave the PSSession running. By doing this, you can run additional commands in the session.

- PSSessions:
 - Represent a persistently running connection on the remote computer
 - Can execute multiple sequences of commands, be disconnected, reconnected, and closed
 - Numerous configuration parameters in the drive WSMAN control idle session time and maximum connections

Disconnected sessions

In Windows PowerShell 3.0 and newer, you also can manually disconnect from sessions. This allows you to close the session in which a PSSession was established, and even shut down the local computer, without disrupting commands running in the PSSession on the remote computer. It is particularly useful for running commands that take a prolonged time to complete, and provides the time and device flexibility that IT professionals need.

Controlling sessions

Every computer has a drive named WSMAN that includes many configuration parameters related to the session. This includes maximum session run time, maximum idle time, maximum number of incoming connections, and maximum number of sessions per administrator. You can explore these configuration parameters by running `dir WSMAN:\localhost\shell`, and you can change them in that same location. You also can control many of the settings through Group Policy.

Creating a PSSession

You use the **New-PSSession** command to create a persistent connection. Notice that the command contains many of the same parameters as **Invoke-Command**, including **-Credential**, **-Port**, and **-UseSSL**. This is because you are creating the identical kind of connection that **Invoke-Command** creates. However, you are leaving this connection running, instead of immediately closing it.

PSSessions do have an idle time-out, after which the remote computer will close them automatically. A closed PSSession differs from a disconnected PSSession, and closed PSSessions cannot be reconnected. In this case, you can only remove the PSSession, and then re-create it.

New-PSSession can accept multiple computer names. This causes it to create multiple PSSession objects. When you run the **New-PSSession** command, it outputs objects representing the newly created PSSessions. You can assign these PSSessions to a variable to make them easier to refer to, and to use in the future.

For example, the following command can then use **\$dc** variable to open a PSSession to LON-DC1 within a script or code block:

```
$dc = New-PSSession -ComputerName LON-DC1
```

- Create sessions by using **New-PSSession**; the command produces a reference to the PSSessions it creates
- Assign PSSessions to variables to make them easier to refer to later

Using a PSSession

You can use a PSSession as soon as you create it. Both the **Invoke-Command** and **Enter-PSSession** commands can accept a PSSession object instead of a computer name. (**Invoke-Command** can accept multiple PSSession objects.) You use the commands' **-Session** parameter for this purpose. When you do this, the commands use the existing PSSession instead of creating a new connection. When your command finishes running or you exit the PSSession, the PSSession remains running and connected, and ready for future use.

- Pass a session object to the **-Session** parameter of **Enter-PSSession** to interactively enter that session
- Alternatively, pass session object(s) to the **-Session** parameter of **Invoke-Command** to run a command against those PSSessions
- The PSSessions remain open and connected after you are finished, leaving them ready for additional use

For example, you can use the following commands to enter a PSSession on **LON-CL1** and then close it:

```
$client = New-PSSession -ComputerName LON-CL1
Enter-PSSession -Session $client
Exit-PSSession
```

Alternatively, you could use the following commands to achieve the same results:

```
$computers = New-PSSession -ComputerName LON-CL1,LON-DC1
Invoke-Command -Session $computers -ScriptBlock { Get-Process }
```

Demonstration: Using PSSessions

In this demonstration, you will see how to create and manage PSSessions.

Demonstration Steps

1. On the **LON-CL1** virtual machine, create a PSSession to **LON-DC1** and store it in a variable.
2. Create sessions to **LON-CL1** and **LON-DC1**, and store them both in a single variable.
3. Display a list of all open PSSessions.
4. Display the status of the **LON-DC1** PSSession.
5. Interactively enter the **LON-DC1** PSSession.
6. Display a list of running processes.
7. Exit the PSSession.
8. Display the status of the **LON-DC1** PSSession.
9. Use remoting to retrieve a list of started services from both **LON-CL1** and **LON-DC1** by using the already-open PSSessions.
10. Close the PSSession to **LON-DC1**.
11. Display a list of all open PSSessions.
12. Close all open PSSessions.

Disconnected sessions

As you have learned, you can disconnect from PSSessions when both the initiating computer and the remote computer are running Windows PowerShell 3.0 and later. Disconnecting is typically a manual process. In some scenarios, Windows PowerShell can automatically place a connection into the **Disconnected** state if the connection is interrupted. However, if you manually close the Windows PowerShell host application, it will not disconnect from sessions but will instead close them. Using disconnected sessions is similar to the following process:

1. Use **New-PSSession** to create the new PSSession. Optionally, use the PSSession to run commands.
2. Run **Disconnect-PSSession** to disconnect from the PSSession. Pass the PSSession object that you want to disconnect from to the command's **-Session** parameter.
3. Optionally, move to another computer and open Windows PowerShell.
4. Run **Get-PSSession** with the **-ComputerName** parameter to obtain a list of your PSSessions running on the specified computer.
5. Use **Connect-PSSession** to reconnect to the desired PSSession.



Note: You cannot see or reconnect to another user's PSSessions on a computer.

• **Disconnect-PSSession** disconnects from a PSSession while leaving Windows PowerShell running:

- Does not happen automatically when you close the host application

• **Get-PSSession -ComputerName** displays a list of your PSSessions on the specified computer:

- You cannot see other users' PSSessions

• **Connect-PSSession** reconnects a PSSession, making it available for use

Demonstration: Disconnected sessions

In this demonstration, you will see how to:

- Create a PSSession.
- Disconnect a PSSession.
- Display PSSessions.
- Reconnect to a PSSession.

Demonstration Steps

1. Create a PSSession to **LON-DC1**. Save the PSSession in a variable.
2. Disconnect from the PSSession.
3. List the PSSessions open in **LON-DC1**.
4. Reconnect to the disconnected PSSession.
5. Verify that your variable contains an active and usable PSSession.
6. Close and remove the PSSession.

Studijní materiály Okskolení

Implicit remoting

One of the ongoing problems in the Windows management space is version mismatch. For example, Windows Server 2016 includes several Windows PowerShell commands. You can install those commands on Windows 10 as part of the RSAT. However, you cannot install the Windows 10 RSAT on a workstation that runs Windows 7.

If you have recently had to rebuild a workstation, you are familiar with another ongoing problem, which is the sheer amount of time that it can take to track down and install administrative tools and Microsoft Management Consoles (MMC) on a computer. Assuming all of them are compatible with your version of Windows, installation alone can take days.

These problems lead administrators to forgo installing tools on workstations, and instead access tools directly on the server through Microsoft Remote Desktop. However, this is not a good solution because it puts the server in the position of having to be a client, while simultaneously providing services to hundreds or thousands of users. The advent of Server Core, which lacks a graphical user interface, was in part to make servers perform better and need fewer updates. However, this also means that they cannot run graphical tools and MMCs.

Implicit remoting brings tools to you

The purpose of implicit remoting is to bring a copy of a server's Windows PowerShell tools to your local computer. In reality, you are not copying the commands at all; you are creating a kind of shortcut, called a *proxy function*, to the server's commands. When you run the commands on your local computer, they implicitly run on the server through remoting. Results are then sent back to you. It is exactly as if you ran everything through **Invoke-Command**, but it is much more convenient. Commands also run more quickly, because commands on the server are naturally co-located with the server's functionality and data.

Using implicit remoting

While implicit remoting was available in Windows PowerShell 2.0, it is much easier to use in Windows PowerShell 3.0 and later. All that is required is to create a session to the server containing the module that you want to use. Then, using **Import-Module** and its **-PSSession** parameter, you import the desired module. The commands in that module, and even its Help files become available in your local Windows PowerShell session.

Using implicit remoting, you have the option of adding a prefix to the noun of the commands that you import. Doing this can make it easier, for example, to have multiple versions of the same commands loaded at the same time without causing a naming collision. For example, if you import both Exchange Server 2013 and Exchange Server 2016 commands, you might add a 2013 and 2016 prefix respectively. This enables you to run both sets of commands. In reality, each would be running on their respective servers, enabling you to run both sets (perhaps in a migration scenario) side-by-side.

Help also works for commands that are running through implicit remoting. However, the Help files are drawn through the same remoting session as the commands themselves. Therefore, the remote computer must have an updated copy of its Help files. This can be a concern on servers, because they might not be used all that frequently, and might not have had **Update-Help** run on them recently to pull down the latest Help files.

- Imports commands from a remote computer to the local one:
 - Imported commands still run on the remote computer through an established remoting session
- Lets you utilize commands without needing to install them
- Help is also drawn from the remote computer

Demonstration: Implicit remoting

In this demonstration, you will see how to use implicit remoting to import and use a module from a remote computer.

Demonstration Steps

1. On **10961C-LON-CL1**, establish a remoting session to **LON-DC1** and save it in a variable.
2. Get a list of modules on **LON-DC1**.
3. Import the Active Directory module from **LON-DC1**, adding the prefix **Rem** to the imported commands' nouns.
4. Display Help for **Get-RemADUser**.
5. Display a list of all domain users.
6. Close the connection to **LON-DC1**.
7. Try running **Get-RemADUser**.

Question: What are some potential operational concerns for PSSessions?

Lab B: Using PSSessions

Scenario

You are an administrator who must perform multiple management tasks on remote computers. In your environment, communications protocols such as RPC are blocked between your local computer and the servers. You plan to use Windows PowerShell remoting, and want to use sessions to both provide persistence and to reduce the setup and cleanup overhead imposed by improvised remoting connections.

Objectives

After completing this lab, you will be able to:

- Create and manage PSSessions.
- Send commands to multiple computers in parallel.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1** and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must follow these steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961C-LON-CL1**.

You should perform these lab steps on the **10961C-LON-CL1** virtual machine.

Exercise 1: Using implicit remoting

Scenario

In this exercise, you will use implicit remoting to import and run commands from a remote computer.

The main tasks for this exercise are as follows:

1. Create a persistent remoting connection to a server.
2. Import and use a module from a server.
3. Close all open remoting connections.

► Task 1: Create a persistent remoting connection to a server

1. On **10961C-LON-CL1**, sign in as **Adatum\Administrator** with the password **Pa55w.rd**.

2. Click the **Start** menu, and then type **powersh**.
3. In the results list, right-click **Windows PowerShell**, and click **Run as administrator**.
4. In the Windows PowerShell command window, create a persistent remoting connection to **LON-DC1**, and save the resulting PSSession object in the variable **\$dc**.
5. Display a list of PSSessions in **\$dc**, and verify their availability.

► **Task 2: Import and use a module from a server**

1. From **LON-CL1**, display a list of Windows PowerShell modules on **LON-DC1**.
2. Locate a module on **LON-DC1** that can work with Server Message Block (SMB) shares.
3. Import the module to your local computer, adding the prefix **DC** to the nouns for each imported command.
4. Display a list of SMB shares on **LON-DC1**.
5. Display a list of SMB shares on the client computer.

► **Task 3: Close all open remoting connections**

- Close all open remoting connections.

Results: After completing this exercise, you should have used implicit remoting to import and run commands from a remote computer.

Exercise 2: Managing multiple computers

Scenario

In this exercise, you will perform several management tasks against multiple computers, relying on PSSessions to provide persistence.

The main tasks for this exercise are as follows:

1. Create PSSessions to two computers.
2. Create a report that displays Windows Firewall rules from two computers.
3. Create and display an HTML report that displays local disk information from two computers.
4. Close all open PSSessions.
5. Prepare for the next module.

► **Task 1: Create PSSessions to two computers**

1. Ensure that you are still signed in to **10961C-LON-CL1** as **Adatum\Administrator** with the password **Pa55w.rd** and that Windows PowerShell is open.
2. Create PSSessions to both **LON-CL1** and **LON-DC1**, and save both PSSession objects in the variable **\$computers**.
3. Verify that the connections in **\$computers** are open and available.

► **Task 2: Create a report that displays Windows Firewall rules from two computers**

1. Find a Windows PowerShell module capable of working with Network Security.

Studijní materiály Okškolení

2. Use a single command line to load the module into memory on both **LON-CL1** and **LON-DC1**.
3. Find a command that can display Windows Firewall rules.
4. Use a single command line to list all enabled firewall rules on both **LON-CL1** and **LON-DC1**.
5. Display only the rule names and the computer name each rule came from.
6. Use a single command line to unload the network security module from memory on both **LON-CL1** and **LON-DC1**.

► **Task 3: Create and display an HTML report that displays local disk information from two computers**

1. As a test, use **Get-WmiObject** to display a list of local hard drives (the **Win32_LogicalDisk** class, filtered to include only those drives with a drive type of 3).
2. Use remoting to run the **Get-WmiObject** command against both **LON-DC1** and **LON-CL1**. Do not add a **-ComputerName** parameter to the **Get-WmiObject** command.



Note: Your report must include each computer's name, each drive's letter, and each drive's free space and total size in bytes.

3. Revise your command from the previous step to produce the desired HTML report.

► **Task 4: Close all open PSSessions**

- Close all open PSSessions.

Results: After completing this exercise, you should have performed several management tasks against multiple computers.

► **Task 5: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Question: What are some of the benefits from using implicit remoting?

Module Review and Takeaways

Best Practices

Always consider the security implications of opening too many remote sessions. The default configuration is tightly secured and provides a good balance of ease-of-use and security/privacy. Make sure that before changing that default configuration, you have explored all the possible ramifications.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
General problems using remoting.	
Remoting will not enable on a client computer.	

Real-world Issues and Scenarios

Many organizations express concerns about remoting's security, and frequently decide to disable it. Remoting is just as secure as Remote Desktop Protocol (RDP), and in some ways is more secure and controllable than many other protocols (such as RPC) that it tries to replace. IT security personnel should take the time to thoroughly understand remoting before deciding to disable it. This is because disabling remoting deprives administrators of a valuable management tool, frequently requiring them to take less-secure workaround measures to restore lost functionality.

There is also concern over the CredSSP protocol that Microsoft describes as an increased security risk. This is because it enables the delegation of credentials to remote computers, and if those computers are compromised then the credential could also be compromised. Only trusted, managed, secured computers should be enabled for CredSSP delegation and even so, the risk is still present. You should use constrained delegation instead of CredSSP.

Studyjní materiály Okškolení

Studijní materiály Okškolení

Module 11

Using background jobs and scheduled jobs

Contents:

Module Overview	11-1
Lesson 1: Using background jobs	11-2
Lesson 2: Using scheduled jobs	11-9
Lab: Using background jobs and scheduled jobs	11-16
Module Review and Takeaways	11-20

Module Overview

In this module, you will learn about the job feature of Windows PowerShell. When you start a task that runs for a long time, your prompt is unavailable until the task completes. Background jobs allow you to get the prompt back while the long-running task continues in memory, or in the *background*, while you continue to do other tasks. When this long-running task is complete, you then can go to the background job to get the results. Jobs are an extension point in Windows PowerShell, and there are several types of jobs available. Each type of job works slightly differently and has different capabilities.



Additional Reading: For more information about jobs, refer to: "About Jobs" at:
<https://aka.ms/b5v2ax>

Objectives

After completing this module, you will be able to:

- Create and manage background jobs.
- Create and manage scheduled jobs.

Lesson 1

Using background jobs

When you run a command as a background job, Windows PowerShell performs the task asynchronously in its own thread separate from the pipeline thread that the command uses. When a command runs as a background job, even if the job takes a long time to complete, you regain control of the command prompt immediately so that you can continue without interruption while the job runs in the background.

In this lesson, you will learn about three types of jobs: local jobs, Windows PowerShell remote jobs, and Common Information Model (CIM)/Windows Management Instrumentation (WMI) jobs. These three jobs form the basis of the Windows PowerShell job system.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and functionality of background jobs.
- Start jobs.
- Manage jobs.
- Retrieve job results.

What are background jobs?

Background jobs are Windows PowerShell commands that run in the background. Each job's command results are stored in memory until you retrieve them.

There are three basic types of background jobs:

- Local jobs. Local jobs run their commands on the local computer. These jobs typically access only local resources. However, you can create a local job whose commands target a remote computer. For example, you could create a local job that includes the following command, whose *-ComputerName* parameter makes it connect to a remote computer:

```
Get-Service -Name * -ComputerName LON-DC1
```

- Remote jobs. These jobs use Windows PowerShell remote to transmit their commands to one or more remote computers. The commands run on those remote computers, and the results are returned to the local computer and stored in memory. Windows PowerShell Help files refer to this kind of job as a *Remote job*.
- CIM and WMI jobs. These jobs use the CIM and WMI repository of management information. The commands run on your computer but can connect to one or more remote computers' repository. Local jobs that use CIM commands use the **Start-Job** command, while WMI and other commands can use the *-AsJob* parameter within a WMI command.

- Run commands in the background
- Store command results in memory for retrieval
- Three basic job types:
 - Local
 - Remoting
 - CIM/WMI
- Each job type has different characteristics

Each type of job has specific characteristics. For example, local and Windows PowerShell remote jobs run in a background Windows PowerShell run space. You can think of them as running in a hidden instance of Windows PowerShell. Other types of jobs might have different characteristics. Also, add-in modules can add more job types to Windows PowerShell, and those job types will have their own characteristics.

Remote jobs are very useful for managing multiple remote computers simultaneously. Because Windows PowerShell remote transmits commands to remote computers, and because the remote computers run those commands by using their own local resources, you can include any command in the job.

Remember that these are not the only types of jobs that Windows PowerShell provides. Modules and other add-ins can create additional job types. Workflow jobs help you automate the long running tasks, or workflows, simultaneously affecting multiple managed computers or devices. Windows PowerShell workflow is intended to be recoverable and robust. Windows PowerShell workflow jobs allow you to restart the target devices and, if the workflow is not finished, the workflow job automatically reconnects to the target device or devices, and continues with the commands in the workflow job.

Note that the interactive Windows PowerShell console workflow jobs are not visible in a non-interactive Windows PowerShell console. This means that scheduling a task to run at machine startup will not find suspended jobs from a non-interactive console. Scheduled tasks cannot find the workflow jobs to resume unless you are logged on to an interactive session.



Additional Reading: For more information on workflow jobs, refer to: "Automatically resuming Windows PowerShell Workflow jobs at logon" at: <https://aka.ms/aj4zzl>

Starting jobs

You start each of the three basic job types in a different way. You can start local jobs, remote jobs, and CIM/WMI jobs. The following section describes the specific methods of calling each job type.

Local jobs

Start local jobs by running **Start-Job**. Provide either the *-ScriptBlock* parameter to specify a single command line or a small number of commands. Provide the *-FilePath* parameter to run an entire script on a background thread.

By default, jobs are given a sequential job identification (ID) number and a default job name. You cannot change the job ID number assigned, but you can use the *-Name* parameter to specify a custom job name. Custom names make it easier to retrieve the job and easier to identify the job in the job list.



Note: At first, job ID numbers might not appear to be sequential. You will learn the reason for this later in this module.

- Local jobs:
`Start-Job -ScriptBlock { Dir }`
- Remoting jobs:
`Invoke-Command -ScriptBlock { Get-Service } -ComputerName LON-DC1 -AsJob`
- CIM/WMI jobs:
`Start-Job -ScriptBlock {Get-CimInstance -ClassName Win32_ComputerSystem}`
`Get-WmiObject -Class Win32_BIOS -ComputerName LON-DC1 -AsJob`

You can specify the *-Credential* parameter to run the job under a different user account. Other parameters allow you to run the command under a specific Windows PowerShell version, in a 32-bit session, and other sessions.

Following are some examples:

```
PS C:\> Start-Job -ScriptBlock { Dir C:\ -Recurse } -Name LocalDirectory
Id     Name          PSJobTypeName   State    HasMoreData  Location
--     --          -----          ----      -----        -----
2     LocalDirectory BackgroundJob  Running   True       localhost
PS C:\> Start-Job -FilePath C:\test.ps1 -Name TestScript
Id     Name          PSJobTypeName   State    HasMoreData  Location
--     --          -----          ----      -----        -----
4     TestScript     BackgroundJob  Running   True       localhost
```

Remote jobs

Start Windows PowerShell remote jobs by running **Invoke-Command**. This is the same command that you would use to send commands to a remote computer. Module 10, “Administering Remote Computers,” covers this command. Add the **-AsJob** parameter to make the command run in the background, and use the **-JobName** parameter to specify a custom job name. All other parameters of **Invoke-Command** are used in the same way. Consider the following example:

```
PS C:\> Invoke-Command -ScriptBlock { Get-EventLog -LogName System -Newest 10 }
-ComputerName LON-DC1,LON-CL1,LON-SVR1 -AsJob -JobName RemoteLogs
Id     Name          PSJobTypeName   State    HasMoreData  Location
--     --          -----          ----      -----        -----
6     RemoteLogs     RemoteJob     Running   True       LON-DC1...
```

 **Note:** Notice that the **-ComputerName** parameter is a parameter of **Invoke-Command**, not of **Get-EventLog**. The parameter causes the local computer to coordinate the Windows PowerShell remote connections to the three computers specified. Each computer receives only the **Get-EventLog** command and runs it locally, returning the results.

Remote jobs are created and managed by the computer where **Invoke-Command** is run. You can refer to that computer as the *initiating computer*. The commands inside the job are transmitted to remote computers. The remote computers execute the commands and return the results to the initiating computer. The initiating computer stores the job’s results in its memory.

CIM and WMI jobs

To use the CIM commands in a job, you must launch the job with **Start-Job**. Consider the following example:

```
PS C:\> Start-Job -ScriptBlock {Get-CimInstance -ClassName Win32_ComputerSystem}
Id     Name          PSJobTypeName   State    HasMoreData  Location  Command
--     --          -----          ----      -----        -----
3     Job3         BackgroundJob  Running   True       localhost Get-CimInstance -Class...
```

You also can run other commands that use CIM, such as **Invoke-CimMethod**, as jobs by using **Start-Job**.

The fact that the CIM commands do not have an **-AsJob** parameter is not an important factor. You just need to remember to use the job commands when you want to run CIM commands as jobs.

Start a WMI job by running **Get-WmiObject**. This is the same command you would use to query WMI instances. Module 6, “Querying management information by using CIM and WMI,” covers this command. Add the **-AsJob** parameter to run the command on a background thread. There is no option to provide a custom job name. The Get-Help information for **Get-WmiObject** states the following for the **-AsJob** parameter.

Note: To use this parameter with remote computers, the local and remote computers must be configured for Windows PowerShell remoting. Additionally, you must start Windows PowerShell by using the "Run as administrator" option in Windows Vista and later versions of Windows. For more information, see [about_Remote_Requirements](#).

WMI jobs do not require Windows PowerShell remoting to be enabled on either the initiating computer or the remote computer. WMI jobs do require that WMI be accessible on the remote computers.

Consider the following example:

```
PS C:\> Get-WmiObject -Class Win32_NTEventLogFile -ComputerName localhost,LON-DC1 -AsJob
Id      Name          PSJobTypeName   State    HasMoreData  Location
--      --          -----          ----    -----       -----
10     Job10         WmiJob        Running   True        Localho...
```

Job objects

Notice that each of the preceding examples produces a job object as their result. The *job object* represents the running job, and you can use it to monitor and manage the job.

Managing jobs

When you start a job, you are given a job object. You can use that object to monitor and manage the job.

Job objects

Each job consists of at least two job objects. The *parent job* is the top-level object, and it represents the entire job, irrespective of the number of computers to which the job connects. The parent job contains one or more *child jobs*. Each child job represents a single computer. A local job contains only one child job. Windows PowerShell remoting jobs and WMI jobs contain one child job for each computer that you specify.

Retrieving jobs

You can list all current jobs by running **Get-Job**. You can list a specified job by adding the *-ID* or *-Name* parameter and specifying the desired job ID or job name. Using the job ID, you can also retrieve child jobs. Consider the following example:

```
PS C:\> Get-Job
Id      Name          PSJobTypeName   State    HasMoreData  Location
--      --          -----          ----    -----       -----
2      LocalDirectory BackgroundJob  Running   True        localhost
4      TestScript      BackgroundJob  Completed  True        localhost
6      RemoteLogs      RemoteJob     Failed    True        LON-DC1...
10     Job10          WmiJob        Failed    False      Localho...
PS C:\> Get-Job -Name TestScript
Id      Name          PSJobTypeName   State    HasMoreData  Location
--      --          -----          ----    -----       -----
4      TestScript      BackgroundJob  Completed  True        localhost
PS C:\> Get-Job -ID 5
Id      Name          PSJobTypeName   State    HasMoreData  Location
--      --          -----          ----    -----       -----
5      Job5          BackgroundJob  Completed  True        localhost
```

• **Get-Job**

- Add *-ID* to retrieve specific job by ID
- Add *-Name* to retrieve specific job by name

• To get a list of child jobs:

```
Get-Job -ID <parent_ID> | Select -ExpandProperty ChildJobs
```

• **Stop-Job**

- **Remove-Job**
- **Wait-Job**

Notice that each job has a state. Parent jobs always display the state of any failed child jobs, even if other child jobs succeeded. In other words, if a parent contains four child jobs, and three of those jobs finished successfully but one failed, the parent job status will be **Failed**.

Listing child jobs

You can list the child jobs of a specified parent job by retrieving the parent job object and expanding its **ChildJobs** property. In Windows PowerShell 3.0 and newer, you can also use the **-IncludeChildJobs** parameter of **Get-Job** to display a job's child jobs.

C:\PS>Get-Job -Name RemoteJobs -IncludeChildJobs					
Id	Name	PSJobTypeName	State	HasMoreData	Location
--	---	-----	-----	-----	-----
7	Job7		Failed	False	LON-DC1
8	Job8		Completed	True	LON-CL1
9	Job9		Failed	False	LON-SVR1

The earlier method used the following example:

PS C:\> Get-Job -Name RemoteLogs Select -ExpandProperty ChildJobs					
Id	Name	PSJobTypeName	State	HasMoreData	Location
--	---	-----	-----	-----	-----
7	Job7		Failed	False	LON-DC1
8	Job8		Completed	True	LON-CL1
9	Job9		Failed	False	LON-SVR1

This technique enables you to discover the job ID numbers and names of the child job objects. Notice that child jobs all have a default name that corresponds with their ID number. The preceding syntax will work in Windows PowerShell 2.0 and newer versions.

Managing jobs

Windows PowerShell includes several commands that you can use to manage jobs. You can pipe one or more jobs to each of these commands, or you can specify jobs by using the **-ID** or **-Name** parameters. Both of those parameters accept multiple values, which means that you can specify a comma-separated list of job ID numbers or names. The job management commands include:

- **Stop-Job**. This stops a job that is running. Use this command to cancel a job that is in an infinite loop or that has run longer than you want.
- **Remove-Job**. This deletes a job object, including any command results that were stored in memory. You should use this command when you are finished working with a job so that the shell can free up memory.
- **Wait-Job**. This is typically used in a script. It pauses script execution until the specified jobs reach the specified state. You can use this in a script to start several jobs, and then make the script wait until those jobs complete before continuing.

The Windows PowerShell process manages remote, WMI, and local jobs. When you close the PowerShell session, Windows PowerShell removes all jobs and their results, and you can no longer access them.

Retrieving job results

When a job runs, Windows PowerShell stores any output its commands produce in memory, starting with the first output that the command produced. You do not have to wait for a command to complete before output becomes available.

The job list shows whether a job has stored results that you haven't yet retrieved. Consider the following example:

PS C:\> Get-Job						
Id	Name	PSJobTypeName	State			
HasMoreData	Location					
--	---	-----	-----			
13	Job13	BackgroundJob	Running	True		localhost

- Use **Receive-Job**
 - Pipe jobs to it to specify jobs
 - Use **-ID** to specify by job ID
 - Use **-Name** to specify by job name

- Add **-Keep** to retain a copy of the results in memory. Otherwise, results are not retained in memory.
- Receiving results from a parent job will receive results from all child jobs.

In this example, job ID 13 is still running, but the **HasMoreData** column indicates that results already have been stored in memory. To receive the results of a job, use the **Receive-Job** command.

By default, job results are removed from memory after they are delivered to you. That means that you can use **Receive-Job** only once per job. Add the **-Keep** parameter to retain a copy of the job results in memory so that you can retrieve them again.

If you retrieve the results of a parent job, you will receive the results from all of its child jobs. If needed, you can also retrieve the results of a single child job or multiple child jobs.

You can retrieve the results of a job that is still running. However, unless you specify **-Keep**, the job object's results will be empty until the job's command adds new output.

For example:

```
Receive-Job -ID 13 -Keep | Format-Table -Property Name,Length
```

Demonstration: Using background jobs

In this demonstration, you will see how to create and manage two types of jobs.

Demonstration Steps

1. Enable Windows PowerShell remote on **LON-CL1**.
2. Start a local job that produces a complete directory listing for **C:**, including subfolders, and name the job **LocalDir**.
3. Start a Windows PowerShell remote job that queries the most recent 100 entries from the Security event log on **LON-CL1** and on **LON-DC1**, and name the job **RemoteLogs**.
4. Display a list of running jobs.
5. Stop the **LocalDir** job.
6. Retrieve the results from the **LocalDir** job.
7. Remove the **LocalDir** job.
8. Display a list of running jobs, and repeat this step every few minutes until the **RemoteLogs** job completes.

9. Display a list of child jobs under the **RemoteLogs** job.
10. Retrieve the results from **LON-DC1**, keeping a copy of the results in memory.
11. Retrieve the results from the **RemoteLogs** parent job.
12. Remove the **RemoteLogs** job.

Question: What are some examples of tasks that you might want to run in the background?

Lesson 2

Using scheduled jobs

In this lesson, you will learn to use scheduled jobs. Similar to background jobs, scheduled jobs run asynchronously in the background. In Windows PowerShell, scheduled jobs are essentially scheduled tasks. Scheduled jobs follow all the same rules for actions, triggers, and other features, and run Windows PowerShell scripts by design.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to run Windows PowerShell scripts as scheduled tasks.
- Create and run a Windows PowerShell script as a scheduled task.
- Explain the purpose and use of scheduled jobs.
- Create job options.
- Create job triggers.
- Create scheduled jobs.
- Retrieve scheduled job results.

Running Windows PowerShell scripts as scheduled tasks

Scheduled jobs are a combination of Windows PowerShell background jobs and Windows Task Scheduler tasks. Similar to background jobs, you define scheduled jobs in Windows PowerShell. Similar to tasks, job results are saved to disk, and scheduled jobs can run even if Windows PowerShell is not running.

Scheduled tasks

Scheduled tasks are part of the Windows core infrastructure components, and other Windows components and products that run on Windows use them extensively. Scheduled tasks are generally simpler than scheduled jobs. The Task Scheduler enables you to schedule the running of almost any program or process, in any security context, triggered by a various system events or a particular date or time.

However, scheduled tasks cannot capture and manipulate task output. Because a scheduled task can run almost anything runnable on a Windows device, it is impossible to anticipate and capture the scheduled task's output. However, because a Windows PowerShell scheduled job is always a Windows PowerShell script, even if that script is used to execute a non-Windows PowerShell program, the system is able to capture output. In this case, a Windows PowerShell object is returned at the end of the script block. A scheduled task consists of:

- Action, which specifies the program to be run.
- Principal, which identifies the context to use to execute an action.

- Windows PowerShell Scheduled Jobs can run in the Task Scheduler
 - Can use all options of the Task Scheduler in a graphical user interface
 - A scheduled task consists of:
 - Action
 - Principal
 - Trigger
 - Additional settings
- Get-Command -Module ScheduledTasks**

- Trigger, which defines the time or system event that determines when the program is to be run.
- Additional settings, which further configure the task and control how the action is run.

You can find the commands that work with scheduled tasks in the **ScheduledTasks** module that is included with Windows 10 and Windows Server 2016. To see the complete list of commands, run the following command:

```
Get-Command -Module ScheduledTasks
```

For example, to check on the various scheduled tasks available, you can run the **Get-ScheduledTask** command. This will list all the available scheduled tasks, whether the tasks are enabled or disabled.

You can get information on a specific task by running the **Get-ScheduledTask** with the *TaskPath* parameter. For best practices, ensure you put the actual path in quotes. You can get further information about a particular task by using the **Get-ScheduledTaskInfo** command. You can then combine these together through a pipeline to get additional information. For example, if you wanted to retrieve detailed information about the Automatic Update task running on your system, you would type the following command:

```
Get-ScheduledTask -TaskPath "\Microsoft\Windows\WindowsUpdate\" | Get-ScheduledTaskInfo
```

You can still create and run scheduled tasks from the Task Scheduler. However, if you are running Windows PowerShell commands or scripts, or Windows tools that do not write their output to a file, and the output is important, a Windows PowerShell scheduled job is the better choice. After that job is in the Task Scheduler, you can manipulate the job further. You can start or stop the job in the Task Scheduler. If you want to create multiple scheduled jobs or tasks locally, or even on remote computers, you can automate their creation and maintenance with the scheduled job or the scheduled task commands.

Adding Windows PowerShell scripts as scheduled jobs in the Task Scheduler can greatly improve your productivity. The script center repository web page contains thousands of scripts that you can use or modify for your specific use, and these scripts are broken down in various categories.



Reference Links: The Microsoft Script Repository is in the Microsoft Script Center, at:
<https://aka.ms/l71no>.

For example, there are hundreds of viable scripts that you can run against the Active Directory Domain Services (AD DS) and other Active Directory services. Some of these scripts can be very useful, such as the script that finds user accounts that have not been used for over 90 days, and then disables them. Doing so can strengthen domain security. You can modify this script to your specific domain, and then create it as a scheduled job. After you configure this task, you can then find and manipulate the job in the Task Scheduler. You can schedule it to run every week, and also report back on what accounts, if any, were disabled.

Demonstration: Using a Windows PowerShell script as a scheduled task

In this demonstration, you will see how to create and run a Windows PowerShell script as a scheduled task.

Demonstration Steps

1. On **LON-DC1**, select a user from the Managers OU in **Active Directory Users and Computers**, and disable the user account.
2. Open the **Task Scheduler**.
3. Create a new task with the following properties:
 - a. Name and Description: **Delete Disabled User from Managers Security Group**
 - b. Security options: **Run whether user is logged on or not** and **Run with highest privileges**
 - c. Trigger settings: **Daily**, set time to five minutes after current time.
 - d. Actions: Program/script text box: **PowerShell.exe -file "E:\Labfiles\Mod11\DeleteDisabledUserManagersGroup.ps1"**
 - e. Settings: **If the task is already running, then the following rule applies: Stop the existing instance**
4. After 5 minutes, review the history of the **Delete Disabled User from Managers Security Group** task.
5. Return to **Active Directory Users and Computers**, and verify that the disabled user account from step 1 is no longer a member of the **Managers** security group.

What are scheduled jobs?

Scheduled jobs are a useful combination of Windows PowerShell background jobs and Windows Task Scheduler tasks. Similar to Task Scheduler tasks, Windows PowerShell scheduled jobs are saved to disk. You can view and manage Windows PowerShell scheduled jobs in the Task Scheduler. From the Task Scheduler, you can enable or disable a task, if needed, or simply run the scheduled job. You can even use the scheduled job as a template for creating other scheduled jobs, or to establish a one-time schedule or periodic schedule for starting the jobs, or to set conditions under which the jobs start again. You can do all of this from the Task Scheduler.

Windows PowerShell saves the results of the scheduled jobs to the disk and creates a running log of the job output. Scheduled jobs have a customized set of commands that allow you to manage them. You can use these commands to create, edit, manage, disable, and reenable scheduled jobs, job triggers, and job options.

Scheduled jobs are a cross between background jobs and Task Scheduler

- Three components:
 - Job definition
 - Job options
 - Job triggers

To see all Scheduled Job commands, use:
Get-Command –Module PSScheduledJob

Studijní materiály Okškolení

To create a scheduled job, use the scheduled job commands. Note that anything created in Task Scheduler is considered a scheduled task even if it is in the **Microsoft\Windows\PowerShell\ScheduledJobs** path in the Task Scheduler. After the scheduled job is created, you can view and manage it in Task Scheduler. In the Task Scheduler, when you select a Windows PowerShell scheduled job, you can find the job triggers on the **Triggers** tab, and you can get the scheduled job options on the **General** and **Conditions** tabs. You can view the job instances that have already been run on the **History** tab. When you change a scheduled job setting in Task Scheduler, the change applies for all future instances of that scheduled job.

You can find the commands that work with scheduled jobs in the **PSScheduledJob** module included with Windows 10 and Windows Server 2016. To see the complete list of commands, run the following command:

```
Get-Command -Module PSScheduledJob
```

Scheduled jobs consist of three components:

- The job itself defines the command that will run.
- Job options define options and execution criteria.
- Job triggers define when the job will run.

You typically create a job option object and a job trigger object, and store those objects in variables. You then use those variables when creating the actual scheduled job.



Note: The **ScheduledTasks** module includes commands that can manage all tasks in the Windows Task Scheduler. This module is included with Windows 10 and Windows Server 2016.

Job options

Use **New-ScheduledJobOption** to create a new job option object. This command has several parameters that let you define options for the job, which include:

- **-HideInTaskScheduler** prevents the job from appearing in the Windows Task Scheduler. If you do not include this option, the final job will appear in the Windows Task Scheduler graphical user interface (GUI).
- **-RunElevated** configures the job to run under elevated permissions.
- **-WakeToRun** will wake the computer when the job is scheduled to run.

- Use **New-ScheduledJobOption** to create an option object
- Parameters correspond to options in the Task Scheduler GUI
- Options include:
 - **-RequireNetwork**
 - **-RunElevated**
 - **-WakeToRun**
 - **-HideInTaskScheduler**

Other parameters allow you to configure jobs that run when the computer is idle, and to configure other options. Many of the parameters correspond to options in the Windows Task Scheduler GUI.

To create a new option object and store it in a variable, use the following command:

```
$opt = New-ScheduledJobOption -RequireNetwork -RunElevated -WakeToRun
```

You do not need to create an option object if you do not want to specify any of its configuration items.

Job triggers

A job trigger defines when a job will run. Each job can have multiple triggers. You create a trigger object by using the **New-JobTrigger** command. There are five basic types of triggers:

- **-Once** specifies a job that runs once. You can also specify a *-RandomDelay*, and you must specify the *-At* parameter to define when the job will run. That parameter accepts a **System.DateTime** object or a string that can be interpreted as a date.
- **-Weekly** specifies a job that runs weekly. You can specify a *-RandomDelay*, and you must specify both the *-At* and *-DaysOfWeek* parameters. *-At* accepts a date and time to define when the job will run. *-DaysOfWeek* accepts one or more days of the week to run the job. You will typically use *-At* to specify only a time, and then *-DaysOfWeek* to define the days for the job.
- **-Daily** specifies a job that runs every day. You must specify *-At* and provide a time when the job will run. You can also specify a *-RandomDelay*.
- **-AtLogOn** specifies a job that runs when the user logs on. This kind of job is similar to a logon script, except that it is defined locally rather than in the domain. You can specify *-User* to limit the user accounts that trigger the job, and *-RandomDelay* to add a random delay.
- **-AtStartUp** is similar to **-AtLogOn**, except that it runs the job when the computer starts. That typically runs the job before a user has an opportunity to log on.

For example, the following command creates a trigger that runs on Mondays and Thursdays every week, at 3:00 PM local time:

```
$trigger = New-JobTrigger -Weekly -DaysOfWeek Monday,Thursday -At '3:00PM'
```

- Use **New-JobTrigger**
- Five basic types of triggers:
 - **-Once**
 - **-Weekly**
 - **-Daily**
 - **-AtLogOn**
 - **-AtStartUp**

Creating a scheduled job

Use **Register-ScheduledJob** to create and register a new scheduled job. Specify any of the following parameters:

- *-Name* is required, and specifies a display name for the job.
- *-ScriptBlock* is required, and specifies the command or commands that the job runs. Or, you might specify *-FilePath* and provide the path and name of a Windows PowerShell script file that the job will run.
- *-Credential* is optional, and specifies the user account that will be used to run the job.
- *-InitializationScript* accepts an optional script block. The command or commands in that script block will execute before the job starts.

- Use **Register-ScheduledJob**
- Creates job definition XML file on disk
- Parameters include:
 - **-Name**
 - **-ScriptBlock** or **-FilePath**
 - **-Credential**
 - **-MaxResultCount**
 - **-ScheduledJobOption** (job option object)
 - **-Trigger** (job trigger object)

- MaxResultCount* is optional, and specifies the maximum number of result sets to store on disk. After this number is reached, the shell will delete older result sets to make room for new ones. The default value for the *-MaxResultCount* is 32.
- ScheduledJobOption* accepts a job option object.
- Trigger* accepts a job trigger object.

To register a new job by using an option object in **\$opt** and a trigger object in **\$trigger**, use the following example:

```
PS C:\> $opt = New-ScheduledJobOption -WakeToRun
PS C:\> $trigger = New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(5)
PS C:\> Register-ScheduledJob -Trigger $trigger -ScheduledJobOption $opt -ScriptBlock {
Dir C:\ } -MaxResultCount 5 -Name "LocalDir"
Id          Name        JobTriggers   Command      Enabled
--          ----        -----        -----      -----
1           LocalDir    1            Dir C:\       True
```

Windows PowerShell registers the resulting job in the Windows Task Scheduler, and creates the job definition on disk. Job definitions are XML files stored in your profile folder, under **\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs**.

You can run **Get-ScheduledJob** to see a list of scheduled jobs on the local computer. If you know a scheduled job's name, you can use **Get-JobTrigger** and the *-Name* parameter to retrieve a list of that job's triggers.

Retrieving job results

Because scheduled jobs can run when Windows PowerShell is not running, results are stored on disk in XML files. If you created the job by using the *-MaxResultCount* parameter, the shell will automatically delete old XML files to make room for new ones so that no more XML files exist than were specified for the *-MaxResultCount*.

After a scheduled job has run, running **Get-Job** in Windows PowerShell displays the results of the scheduled job as a job object.

Consider the following example:

```
PS C:\> Get-Job
Id      Name      PSJobTypeName     State      HasMoreData      Location      Command
--      ----      -----          -----      -----          -----          -----
6       LocalDir  PSScheduledJob  Completed  True           Localhost    Dir C:\
```

- Run **Get-Job** to see a list of **PSScheduledJob** jobs
 - Each represents an execution of the scheduled job
 - Provides access to the job results
- Run **Receive-Job** to retrieve results
- Run **Remove-Job** to delete results and the job object

You can then use **Receive-Job** to receive the results of a scheduled job. If you do not specify **-Keep**, you can receive the results of a job only once per Windows PowerShell session. However, because the results are stored on disk, you can open a new Windows PowerShell session and receive the results again. For example:

```
PS C:\> Receive-Job -id 6 -Keep
Directory: C:\

Mode          LastWriteTime    Length Name
----          -----        ----
d---          7/26/2012 12:33 AM      PerfLogs
d-r--         11/28/2012 1:54 PM       Program Files
d-r--         12/28/2012 2:22 PM       Program Files (x86)
d----         11/16/2012 9:33 AM      reports
d----         9/18/2012 7:28 AM       Review
d----         1/5/2013  7:49 AM       scr
d----         1/5/2013  7:50 AM       scrx
d-r--         9/15/2012 8:16 AM      Users
d----         12/19/2012 3:24 AM      Windows
-a---        1/1/2013  9:39 AM    2892628 EventReport.html
-a---        1/2/2013  12:37 PM      82 Get-DiskInfo.ps1
-a---        12/30/2012 12:33 PM      246 test.ps1
```

Each time the scheduled job runs, a new job object is created to represent the results of the most recent job execution. You can use **Remove-Job** to remove a job as shown in the following example. Doing so deletes the results file from disk. For example:

```
PS C:\> Get-Job -id 6 | Remove-Job
```

Demonstration: Using scheduled jobs

In this demonstration, you will see how to create, run, and retrieve the results from a scheduled job.

Demonstration Steps

1. On **LON-CL1**, in Windows PowerShell, remove all jobs.
2. Create a job trigger that will run a job once in 2 minutes. Store the trigger in the variable **\$trigger**.
3. Using **\$trigger**, create a job named **DemoJob** that retrieves all entries from the Application event log on the local computer.
4. Display the triggers for the scheduled job. Notice the time that the job is scheduled to run.
5. Wait for the job to run. While waiting for the job to run, display a list of scheduled jobs.
6. After the job runs, display a list of jobs.
7. Receive the results of the **DemoJob** job.
8. Remove the **DemoJob** job results.

Question: Why would you use **Register-ScheduledJob** from the **PSScheduledJob** module instead of a command in the **ScheduledTasks** module?

Studijní materiály Okškolení

Lab: Using background jobs and scheduled jobs

Scenario

Background jobs provide a useful way to run multiple commands at the same time, and to run long-running commands in the background. In this lab, you will learn to create and manage two of the three basic kinds of jobs.

In this lab, you will create and configured two scheduled jobs. You will also create a Scheduled Task using a Windows PowerShell script that looks for and removes disabled accounts from a certain security group.

Objectives

After completing this lab, you will be able to:

- Create and manage a remote job.
- Create and manage a scheduled job and a scheduled task.

Lab Setup

Estimated Time: **30 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **ADATUM\Administrator**

Password: **Pa55w.rd**

The changes that you make during this lab will be lost if you revert your VMs at another time during class.

For this lab, you will use the available VM environment. Before you begin the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Hyper-V Manager, click **10961C-LON-DC1**, and in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **ADATUM**
5. Repeat steps 2-4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.
6. Perform the lab steps on the **10961C-LON-CL1** virtual machine.

Exercise 1: Starting and managing jobs

Scenario

In this exercise, you will start jobs using two of the basic job types.

The main tasks for this exercise are as follows:

1. Start a Windows PowerShell remote job.
2. Start a local job.
3. Review job status.

4. Stop a job.
5. Retrieve job results.

► **Task 1: Start a Windows PowerShell remote job**

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. Start a Windows PowerShell remoting job that retrieves a list of physical network adapters from **LON-DC1** and **LON-SVR1**. Name the job **RemoteNetAdapt**.
3. Start a Windows PowerShell remote job that retrieves a list of Server Message Block (SMB) shares from **LON-DC1** and **LON-SVR1**. Name the job **RemoteShares**.
4. Start a Windows PowerShell remote job that retrieves all instances of the **Win32_Volume** CIM class from every computer in Active Directory Domain Services. Name the job **RemoteDisks**. Because some domain computers may not be started, some child jobs might fail.

► **Task 2: Start a local job**

1. Start a local job that retrieves all entries from the Security event log. Name the job **LocalSecurity**.
2. Using the range operator (..) and **ForEach-Object**, start a local job that produces 100 directory listings of drive C, including subfolders. Name the job **LocalDir**. Proceed to the next task while this job is still running.

► **Task 3: Review job status**

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. Display a list of running jobs.
3. Display a list of running jobs whose names start with **remote**.

► **Task 4: Stop a job**

- Force the **LocalDir** job to stop.

► **Task 5: Retrieve job results**

1. Wait for all remaining jobs to either complete or fail.
2. Receive the results of the **RemoteNetAdapt** job.
3. Use a single command line to receive the results from **LON-DC1** for the **RemoteDisks** job.

You must start by querying the parent job, and then expanding its **ChildJob** property. Filter the child jobs so that just the **LON-DC1** job remains, and then receive the results from that job. You will use a total of four commands to complete this step.

 **Note:** You will receive an error on the **LON-CL1** data collection due to Windows PowerShell Remoting not being turned on. This is expected.

Results: After completing this exercise, you will have started jobs using two of the basic job types.

Exercise 2: Creating a scheduled job

Scenario

In this exercise, you will create and run a scheduled job and retrieve the job's results. You will then create and run a scheduled task from a Windows PowerShell script that removes a disabled user from a security group in the AD DS.

The main tasks for this exercise are as follows:

1. Create job options.
2. Create a job trigger.
3. Create a scheduled job and retrieve results.
4. Use a Windows PowerShell script as a scheduled task.
5. Prepare for the next module.

► Task 1: Create job options

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. Create a job option object and store it in **\$option**. Configure the job object as follows:
 - o The job will wake the computer to run.
 - o The job will run under elevated permissions.

► Task 2: Create a job trigger

- Create a job trigger object and store it in **\$trigger1**. Configure the trigger as follows:
 - o The job will run once in 5 minutes. Use **Get-Date** and a method of the resulting **DateTime** object to calculate 5 minutes from now.

► Task 3: Create a scheduled job and retrieve results

1. Using **\$option** and **\$trigger1**, create a new scheduled job that has the following attributes:
 - a. The job action retrieves all entries from the Security event log.
 - b. The job name is **LocalSecurityLog**.
 - c. The maximum number of job results is five.
2. Display a list of job triggers, including time, for the **LocalSecurityLog** scheduled job.
3. Wait until the time shown in step 2 has passed.
4. Display a list of jobs.
5. Receive the job results for **LocalSecurityLog**.

► Task 4: Use a Windows PowerShell script as a scheduled task

1. On **LON-DC1**, open **Active Directory Users and Computers**.
2. Select a user from the **Managers** organizational unit (OU) in **Active Directory Users and Computers**, and disable the user account.
3. Open the **Task Scheduler**.
4. Create a new task with the following properties:
 - a. Name and Description: **Delete Disabled User from Managers Security Group**
 - b. Security options: **Run whether user is logged on or not** and **Run with highest privileges**

- c. Trigger settings: **Daily**, set time to five minutes after current time.
 - d. Actions: Program/script text box: **PowerShell.exe**
 - e. In the **Add arguments (optional)**: type **-ExecutionPolicy Bypass E:\Labfiles\Mod11\DeleteDisabledUserManagersGroup.ps1**
 - f. Settings: **If the task is already running, then the following rule applies: Stop the existing instance**
5. After 5 minutes, view the history of the **Delete Disabled User from Managers Security Group** task.
 6. Return to **Active Directory Users and Computers** and verify that the disabled user account from step 1 is no longer a member of the **Managers** security group.

Results: After completing this exercise, you will have created and run a scheduled job, and retrieved the job's results. You will also have a scheduled task that uses a Windows PowerShell script to remove disabled users from a security group.

► Task 5: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: **Get-CIMInstance** does not have an **-AsJob** parameter. Why? How would you use **Get-CimInstance** in a job?

Question: Is it possible to create a scheduled job without creating a job option object?

Module Review and Takeaways

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The ScheduledTasks module is not available.	

Review Question

Question: What is the main difference between a background job and a scheduled job?

Real-world Issues and Scenarios

Remember that scheduled jobs are defined, stored, and managed locally. There is no option for centralized scheduled job management. For that reason, you should be careful to document job definitions. In large environments, you might prefer to use a centralized job management solution, such as Microsoft System Center Orchestrator.

Module 12

Using advanced Windows PowerShell techniques

Contents:

Module Overview	12-1
Lesson 1: Creating profile scripts	12-2
Lesson 2: Using advanced techniques	12-5
Lab: Practicing advanced techniques	12-17
Module Review and Takeaways	12-22

Module Overview

Windows PowerShell provides many advanced techniques and features that you might not be aware of yet. This module introduces you to some of these advanced techniques and features, including profile scripts, regular expressions, and the format operator. Many of these techniques and features extend the functionality that you have learned about in previous modules. Some of these techniques are new to Windows PowerShell 5.0 and provide additional capabilities.

Objectives

After completing this module, you will be able to:

- Create and manage profile scripts.
- Use advanced Windows PowerShell techniques to work with data.

Lesson 1

Creating profile scripts

Profile scripts are a Windows PowerShell feature that you can use to specify a list of commands that run every time you open a new Windows PowerShell session. You can use profile scripts to customize your own Windows PowerShell environment or the Windows PowerShell environment for all users on a computer.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of a profile script.
- List the correct locations for profile script storage.
- Describe security concerns for the profile script feature.
- Create a profile script.

What is a profile script?

A profile script is a Windows PowerShell script that the host application automatically loads and runs each time it starts. Host applications might implement support for multiple profile scripts in various locations. This enables the host applications to load and run several scripts each time they start.

Profile scripts are a feature of Windows PowerShell host applications and not of the Windows PowerShell engine. This means that the host application is responsible for processing profile scripts. The console host (Windows PowerShell prompt) and the Windows PowerShell Integrated Scripting Environment (Windows PowerShell ISE) implement profile scripts. Other host applications might not use profile scripts.

Profile scripts can contain any commands that you want. However, administrators typically use them to customize their Windows PowerShell environments. In Windows 7 and Windows Server 2008 R2, modules did not load automatically when you referenced a cmdlet, and you could automate the loading of modules in a profile script. In newer versions of Windows, it is more common to use a profile script to define variables or a PSDrive.

Profile scripts are specific to the computer that they are stored on. If you configure aliases or variables in a profile script, then you need to remember that those items are only local when you create scripts. A script that uses those items will not run properly on a computer that does not have the same customizations.

 **Note:** When you connect to a remote computer by using Windows PowerShell remoting, no profile script runs on the remote computer.

- Profile scripts:
 - Are processed by the host application
 - Load automatically each time the host application starts
- Use profile scripts to customize the Windows PowerShell environment
 - Create aliases
 - Set variables
- Profile script customizations are specific to the local computer

Profile script locations

The console and Windows PowerShell ISE host application are each capable of loading up to four profile scripts when a new session starts. For example, when you open a Windows PowerShell prompt, the host application processes up to four profile scripts to customize the environment. When you open a second Windows PowerShell prompt, that host application processes the same profile scripts for a separate prompt.

The processing of profile scripts is dependent on their locations and names. The location of the script defines whether the profile script loads for all users or just the current user. The name of the script defines which host applications use it.

The following table lists the locations that you can use for profile scripts.

Users affected	Location
All users	\$pshome (C:\Windows\System32\WindowsPowerShell\v1.0)
Current user	\$home\Documents\WindowsPowerShell

The following table lists the file names that you can use for profile scripts.

Host application	File name
All hosts	Profile.ps1
Console	Microsoft.PowerShell_profile.ps1
Windows PowerShell ISE	Microsoft.PowerShellISE_profile.ps1

Profile script security concerns

Profile scripts are simple text files that you can easily modify. Malware can also modify profile scripts and use them as a method for running unwanted software on your computer. Because you often have elevated privileges when running Windows PowerShell scripts from a prompt, this is a significant risk. You need to consider how you can mitigate this risk.

A profile script for all users is at less risk than a profile script for a single user. Access to the **\$pshome** folder is restricted to administrative users. Even if you are signed in as a user with administrative permissions, User Account Control prevents access to **\$pshome** unless you specifically choose to elevate privileges. You can create or modify a profile script located in your user profile without elevating privileges.

Users affected	Profile script locations
All users	\$pshome (C:\Windows\System32\WindowsPowerShell\v1.0)
Current user	\$home\Documents\WindowsPowerShell
Host application	Profile script file names
All hosts	Profile.ps1
Console	Microsoft.PowerShell_profile.ps1
Windows PowerShell ISE	Microsoft.PowerShellISE_profile.ps1

- Profile scripts:
 - Can be modified by malware
 - Are at less risk in **\$pshome**

- To mitigate against profile script modification:
 - Install antimalware software and keep definitions up to date
 - Install Windows updates in a timely manner
 - Use an **AllSigned** execution policy
 - Use a separate administrative account and **Run as administrator**

As a defense against malware, you should:

- Install antimalware software and keep the definitions up to date.
- Install Windows updates in a timely manner.

To mitigate specifically against unauthorized modifications of profile scripts, you can use the **AllSigned** execution policy. When the execution policy is **AllSigned**, the profile scripts must have a digital signature, and any unauthorized modifications to the profile script render the digital signature invalid.

Using a separate administrative account also provides some protection. When you use **Run as administrator** to open a Windows PowerShell prompt as an administrative user, it uses the profile scripts of that user instead of the currently signed-in user. If malware has modified the profile script of the currently signed-in user, it will not affect the Windows PowerShell prompt that you opened by using **Run as administrator**.

Demonstration: Creating a profile script

In this demonstration, you will see how to create a profile script.

Demonstration Steps

1. On **LON-CL1**, create a **WindowsPowerShell** folder in the **Documents** folder of your user profile.
2. In the **WindowsPowerShell** folder, create a file named **Profile.ps1**.
3. Edit **Profile.ps1**, and then add the line **\$servers="LON-DC1" "LON-SVR1"**.
4. Open a Windows PowerShell prompt and verify that **\$servers** has the correct value.

Question: Why is the location in which you store a user profile script important?

Lesson 2

Using advanced techniques

Windows PowerShell has many advanced techniques that you might use only in specific scenarios. Some of these advanced techniques—such as using regular expressions, the format operator, and setting New Technology File System (NTFS) permissions—are more complex than most tasks you perform by using Windows PowerShell. There are additional techniques that you might use less often. These techniques include working with secure strings for passwords and running external commands.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use passwords and secure strings.
- Explain how to use array operators.
- Explain how to use regular expressions.
- Use regular expressions.
- Describe how to use the format operator.
- Use the format operator.
- Explain how to run external commands.
- Describe how to configure NTFS permissions.
- Set NTFS permissions.
- Explain how to log the use of Windows PowerShell.

Passwords and secure strings

Windows PowerShell cmdlets and credentials that work with passwords use secure strings to store passwords. A secure string is encrypted in memory and only the user that encrypted the string can decrypt it. This prevents the other processes that are running on the computer from gaining access to the secure string.

You may need to convert plain text to a secure string. For example, if a comma-separated value (CSV) file contains default passwords for new user accounts, then you need to convert each password to a secure string so that you can set the password by using the **New-ADUser** cmdlet.

When you convert a string in memory to a secure string, it is inherently risky because the string exists in memory as unencrypted text before it is converted. Consequently, you need to force Windows PowerShell to allow the conversion.

- A secure string is encrypted in memory
- To convert a string to a secure string:
 - `$string | ConvertTo-SecureString -AsPlainText -Force`
- To accept user input as a secure string:
 - `Read-Host "Enter password" -AsSecureString`
- To store and retrieve secure strings on disk use:
 - `Export-Clixml`
 - `Import-Clixml`

To convert a string to a secure string, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$securePass = $string | ConvertTo-SecureString -AsPlainText -Force
```

If you want to store user input as a secure string, you can use the **-AsSecureString** parameter with **Read-Host**. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$securePass = Read-Host "Enter password" -AsSecureString
```

You can use **ConvertFrom-SecureString** to extract text that can be stored to a file for later use. However, it is preferred to use **Export-Clixml** and **Import-Clixml** to store and retrieve secure strings.

Array operators

You can use specific array operators to identify whether items exist in an array. For example, you can use the **-in**, **-notin**, **-contains**, or **-notcontains** operators to find whether a value exists in an array, without having to evaluate each item in the array.

The **-in**, **notin**, **-contains**, and **-notcontains** operators do not perform partial matches or pattern-based matches. The match with the item must be exact.

To evaluate whether an item exists in an array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
"item" -in $array
```

- The **-in** and **-contains** operators do not perform partial matches
- To evaluate whether an item is in an array:
 - **"item" -in \$array**
- To evaluate whether an item is not in an array:
 - **"item" -notin \$array**
- To evaluate whether an array contains an item:
 - **\$array -contains "item"**
- To evaluate whether an array does not contain an item:
 - **\$array -notcontains "item"**

To evaluate whether an item does not exist in an array, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
"item" -notin $array
```

To evaluate whether an array contains an item, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$array -contains "item"
```

To evaluate whether an array does not contain an item, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
$array -notcontains "item"
```

Regular expressions

You can use the **-match** operator to perform more complex pattern matching than is possible with the **-like** operator. The **-match** operator uses regular expressions to identify complex patterns. The syntax for regular expressions is very flexible, but it can seem complex at first. The asterisk (*) and question mark (?) symbols have different meanings in regular expressions.

To evaluate if a simple string exists within a larger string, at the Windows PowerShell prompt, type the following command, and then press Enter:

```
"Large string" -match "large"
```

The **-match** operator uses regular expressions
• "large string" -match "large"

Symbol	Description	Symbol	Description
.	One of any character	?	Zero or one of any character
+	One or more of the preceding character	*	Zero or one of the preceding character
\w	Any word character	\W	Any non-word character
\s	Any space character	\S	Any non-space character
\d	Any digit	\D	Any non-digit
[abc]	Any specified character	[a-f]	Any character in range
^	Start of string	\$	End of string
{1,3}	Minimum and maximum of previous character		

The following table lists the patterns and symbols that you can use to build regular expressions.

Symbol or pattern	Description
.	One of any character. For example, "fin" and "fun" match "f.n".
?	Zero or one of any character. For example, "fun" and "fn" match "f?n".
+	One or more of the preceding character. For example, "mood" and "mod" match "mo+d".
*	Zero or more of the preceding character. For example, "mood" and "md" match "mo*d".
\w	Any word character (letter or number). For example, "x" and "1" match "\w".
\s	Any space character (such as space or tab). For example, " " matches "\s".
\d	Any digit character (number). For example, "1" and "5" match "\d".
\W	Any character that is not a word. For example, " " matches "\W".
\S	Any character that is not a space. For example, "x" and "@" match "\S".
\D	Any character that is not a digit. For example, "x" and " " match "\D".
[bcf]	Any specified character. For example, "bad" and "cad" match "[bcd]ad".
[b-f]	Any character in the specified range. For example, "bad" and "cad" match "[b-f]ad".
{1,3}	Minimum and maximum number of previous character type. For example, "123" and "4" match "\d{1,3}".
^	Start of string. For example, "first word" matches "^first" but does not match "word".
\$	End of string. For example, "last word" matches "word\$" but does not match "last\$".



Note: If you need to match one of the special characters for a regular expression, you can use "\" as an escape character. For example, "\\$" allows you to match "\$", instead of \$ identifying the end of the string.

Demonstration: Using regular expressions

In this demonstration, you will see how to use regular expressions.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. To see matching of a substring, type the following command, and then press Enter:
`"A Large string" -match "large"`
3. To see that a dot matches one character, type the following command, and then press Enter:
`"LON-DC1" -match "LON-DC."`
4. To see that a dot does not match zero characters, type the following command, and then press Enter:
`"LON-DC" -match "LON-DC."`
5. To see that a question mark matches one character, type the following command, and then press Enter:
`"LON-DC1" -match "LON-DC?"`
6. To see that a question mark matches zero characters, type the following command, and then press Enter:
`"LON-DC" -match "LON-DC?"`
7. To see that a plus matches one instance of the preceding character, type the following command, and then press Enter:
`"LON-DC1" -match "LON-DC+1"`
8. To see that a plus matches multiple instances of the preceding character, type the following command, and then press Enter:
`"LON-DCCCC1" -match "LON-DC+1"`
9. To see that a plus does not match zero instances of the preceding character, type the following command, and then press Enter:
`"LON-D1" -match "LON-DC+1"`
10. To see that an asterisk matches one instance of the preceding character, type the following command, and then press Enter:
`"LON-DC1" -match "LON-DC*1"`

11. To see that an asterisk matches multiple instances of the preceding character, type the following command, and then press Enter:

```
"LON-DCCCC1" -match "LON-DC*1"
```

12. To see that an asterisk matches zero instances of the preceding character, type the following command, and then press Enter:

```
"LON-D1" -match "LON-DC*1"
```

13. To see that "\w" matches word characters, type the following command, and then press Enter:

```
"LON-DC1" -match "LON-DC\w"
```

14. To see that "\s" matches space characters and not digits, type the following command, and then press Enter:

```
"LON-DC1" -match "LON-DC\s"
```

15. To see that "\d" matches digit characters, type the following command, and then press Enter:

```
"LON-DC1" -match "LON-DC\d"
```

16. To see matching minimum and maximum instances of a character, type the following command, and then press Enter:

```
"1-1-1" -match "\d{1,3}-\d{1,3}-\d{1,3}"
```

17. To see matching minimum and maximum instances of a character, type the following command, and then press Enter:

```
"1-123-1" -match "\d{1,3}-\d{1,3}-\d{1,3}"
```

18. To see matching minimum and maximum instances of a character fail, type the following command, and then press Enter:

```
"1-1234-1" -match "\d{1,3}-\d{1,3}-\d{1,3}"
```

19. To see matching of a substring without defining start and end, type the following command, and then press Enter:

```
"1234" -match "\d{1,3}"
```

20. To see matching of a string with the start and end defined fail, type the following command, and then press Enter:

```
"1234" -match "^\\d{1,3}$"
```

The format operator

To perform more precise formatting of the text that displays on the screen, you can use the **format (-f)** operator to provide additional formatting information. You can use the format operator to specify formatting, such as:

- Right or left alignment.
- Number of decimal places.
- Custom date formats.

When using the format operator, you replace the content to be displayed with a set of braces, by using the following syntax:

```
{index,alignment:format}
```

- Use the format operator to perform precise formatting of text
- Syntax:
• {index,alignment:format}
- Examples:
• "File size: {0,10:f2}" -f 1.0782345
• "{0,-20} {1,10}" -f "File Name","File Size"

Format string	Description
c	Currency
f	Fixed point
n	Number
x	Hexadecimal
hh	Hours
mm	Minutes
ss	Seconds

Index numbers start at 0 and the first value listed after the **-f** operator is inserted at index 0. The alignment value is optional. For right alignment, the value is positive and for left alignment, the value is negative. For example, -10 allows for ten characters and left aligns the value.

The format string is optional. The format string describes what the value should look like. This is where you can specify a number of decimal places or custom date formatting.

The following example has only a single item being formatted with an index number of 0. Ten spaces are allocated for the value, and it is right aligned. The format string **f2** defines a fixed decimal with two decimal places. The value of 1.0782345 will round up to 1.08:

```
"File size: {0,10:f2}" -f 1.0782345
```

If you use **-f** as a parameter with **Write-Host**, it will be interpreted as the *-ForegroundColor* parameter. To use the **-f** operator with the **Write-Host** cmdlet, you need to enclose the text being displayed in parentheses. This forces it to be evaluated before it is processed by **Write-Host**. For example:

```
Write-Host ("File size: {0,10:f2}" -f 1.0782345)
```

The following example creates column headers. The first column for file names is left aligned and the second for file size is right aligned:

```
"{0,-20} {1,10}" -f "File Name","File Size"
```

The following table lists some of the commonly used format strings.

Format string	Description
c	Currency format that adds the appropriate currency indicator. You can specify the number of decimal places.
f	Fixed-point format that you can use to specify a number of decimal places.
n	Number format that you can use to specify a number of decimal places. This format adds a separator for thousands.
x	Hexadecimal format.

Format string	Description
hh	Hours from a date value.
mm	Minutes from a date value.
ss	Seconds from a date value.

 **Additional Reading:** For additional information on format strings, refer to "Formatting Types in .NET" at: <https://aka.ms/jyvw6j>

Demonstration: Using the format operator

In this demonstration, you will see how to use the format operator.

Demonstration Steps

1. Open a Windows PowerShell prompt.
2. To see index numbers for the format operator, type the following command, and then press Enter:

```
"File name: {0} File size: {1}" -f "Test.txt",20023.34587
```

3. To see index numbers for the format operator, type the following command, and then press Enter:

```
"File name: {1} File size: {0}" -f "Test.txt",20023.34587
```

4. To see alignment, type the following command, and then press Enter:

```
"File name: {0,-15} File size: {1,12}" -f "Test.txt",20023.34587
```

5. To see the fixed-point format string, type the following command, and then press Enter:

```
"File name: {0,-15} File size: {1,12:f2}" -f "Test.txt",20023.34587
```

6. To see the number format string, type the following command, and then press Enter:

```
"File name: {0,-15} File size: {1,12:n2}" -f "Test.txt",20023.34587
```

7. To see the currency format string, type the following command, and then press Enter:

```
"Expense name: {0,-15} Cost: {1,12:c}" -f "Dinner",53.25
```

8. To see a custom time format, type the following command, and then press Enter:

```
"{0,2:hh}:{0,2:mm}" -f (Get-Date)
```

Studijní materiály Okškolení

Running external commands

Windows PowerShell supports the running of external commands. External commands are executable files that perform a task. For example, **ping** is an external command that you can run at a Windows PowerShell prompt.

Most of the time, when you run an external command, Windows PowerShell creates an instance of **cmd.exe**, runs the command, and captures the output. The output is returned to the Windows PowerShell pipeline as a collection of string objects.

If the command line to run the external command uses reserved characters, then Windows PowerShell might not interpret it properly. For example, **bcdedit.exe** uses braces in its syntax, which causes problems with interpretation. To avoid problems with incorrectly interpreted characters, you can prevent Windows PowerShell from interpreting content after the external command name.

The following example uses **icacls.exe** to set permissions on a folder but does not run properly at a Windows PowerShell prompt:

```
icacls.exe C:\Test /Grant Users(F)
```

You can use the stop parsing symbol (--) to prevent interpretation of the parameters. All text after the stop parsing symbol is treated as literal and Windows PowerShell does not evaluate it. The following example shows how to use the stop parsing symbol:

```
icacls.exe --% C:\Test /Grant Users(F)
```



Note: You can also use the backtick character (`) to prevent interpretation of a single character. However, it is simpler to use --%.



Additional Reading: For additional information about the stop parsing symbol, refer to: "About Parsing" at: <https://aka.ms/vhxi7r>

Working with NTFS permissions

To view and manipulate NTFS permissions, Windows PowerShell includes the **Get-Acl** and **Set-Acl** cmdlets. Unfortunately, these cmdlets aren't easy to work with. When you want to manipulate NTFS permissions, you might decide to continue using external commands such as **icacls.exe** or modules that provide additional cmdlets for manipulating NTFS permissions.

When you use the **Get-Acl** cmdlet for a file or folder, it provides basic access information in the default output. The **Access** column identifies users and groups that have access and their levels of access. However, this does not indicate whether the permissions are set directly on the file or folder or whether they are inherited.

- To manipulate NTFS permissions, you can use:
 - **Get-Acl**
 - **Set-Acl**
- To view detailed access information from an ACL stored in a variable:
 - **\$acl.Access**
- To modify the ACL:
 1. Use **Get-Acl** to put the existing ACL into a variable
 2. Create a new rule and place it in a variable
 3. Add the new rule to the ACL in the variable
 4. Use **Set-Acl** to configure the ACL on the file or folder as the modified variable

 **Note:** The Access column that displays in the default output for the **Get-Acl** cmdlet is actually the **AccessToString** property of the access control list (ACL). The **Access** property of the ACL provides information that is more detailed.

To identify whether permissions are inherited, use the following syntax to view detailed permission information:

```
$acl = Get-Acl C:\test\file.txt
$acl.Access
```

You can use the **Get-Member** cmdlet to view all of the methods available for working with an ACL. The syntax for each method is relatively complex, and you will need to research how to use each method.

To modify security information for a file or folder, you need to perform a multistep process. To add a new access rule to a file or folder, perform the following steps:

1. Use **Get-Acl** to put the existing ACL into a variable.
2. Create a new rule and place it in a variable.
3. Add the new rule to the ACL in the variable.
4. Use **Set-Acl** to configure the ACL on the file or folder as the modified variable.

 **Note:** The **NTFSecurity** module, which a Microsoft employee created, provides simplified cmdlets for working with NTFS permissions. You can locate this module by using **Find-Module**.

 **Additional Reading:** For more information on permissions, refer to: "NTFS Security Tutorial 1 – Getting, adding and removing permissions" at: <https://aka.ms/hbumre>

 **Additional Reading:** For more information on NTFS inheritance, refer to: "NTFS Security Tutorial 2 – Managing NTFS Inheritance and Using Privileges" at: <https://aka.ms/th8i3x>

Demonstration: Setting NTFS permissions

In this demonstration, you will see how to set NTFS permissions.

Demonstration Steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. To create a new folder, type the following command, and then press Enter:

```
New-Item C:\test -ItemType Directory
```
3. To put the ACL for the folder in **\$acl**, type the following command, and then press Enter:

```
$acl = Get-Acl C:\test
```
4. To view the contents of **\$acl**, type the following command, and then press Enter:

```
$acl
```
5. To view a summary of access rules, type the following command, and then press Enter:

```
$acl.AccessToString
```
6. To view detailed access rules, type the following command, and then press Enter:

```
$acl.Access
```
7. To view the properties and methods for an ACL, type the following command, and then press Enter:

```
$acl | Get-Member
```
8. To disable inheritance and clear inherited permissions, type the following command, and then press Enter:

```
$acl.SetAccessRuleProtection($true,$false)
```
9. To create a new access rule for **Administrators**, type the following command, and then press Enter:

```
$rule = New-Object  
System.Security.AccessControl.FileSystemAccessRule("Administrators","FullControl",  
"ContainerInherit, ObjectInherit", "None", "Allow")
```
10. To add the access rule to the ACL, type the following command, and then press Enter:

```
$acl.AddAccessRule($rule)
```
11. To apply the ACL to **C:\Test**, type the following command, and then press Enter:

```
Set-Acl C:\Test -AclObject $acl
```
12. To verify that the permissions were modified, type the following command, and then press Enter:

```
Get-Acl C:\Test | FL
```

Logging activity

You can use several methods to monitor the use of Windows PowerShell on a computer. Module logging and script block logging write events to the **PowerShell Operational** log. Transcription logs record all commands and results during a session.

Module logging

You can enable module logging by using a Group Policy Object (GPO). The **Turn on Module Logging**

Logging setting is located in **\Policies\Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell**.

When you enable this setting, you need to specify which modules you are enabling for logging. Use * to enable logging for all modules.

 **Note:** The **Turn on Module Logging** setting is also available in the **User Configuration** section of a GPO. If you configure the setting in both locations, the **Computer Configuration** has priority.

The events that module logging generates are information events with the event ID 4103. The description of the event identifies the command that ran.

Script block logging

Script block logging records entire script blocks as they run. This is easier to interpret than individual commands. This feature is available in Windows PowerShell 4.0 and newer. Windows PowerShell 5.0 logs suspicious script blocks automatically, even when you have not enabled script block logging.

 **Note:** Script block logging is available in Windows PowerShell 4.0 only after KB 3000850 is installed.

You can enable script block logging by using a GPO. The **Turn on PowerShell Script Block Logging** setting is located in **\Policies\Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell**. In this setting, you also have the option to enable logging of the time when the processing of each script block starts and stops. You can also configure this setting in the **User Configuration** section of the GPO.

The events that script block logging generates are information events with the event ID 4104.

Transcription

Older versions of Windows PowerShell supported using **Start-Transcript** to record the commands and their results in Windows PowerShell sessions. This was an effective way to log your own actions, but it required you to remember to run the **Start-Transcript** cmdlet. Alternatively, you could add **Start-Transcript** to a profile script to have it run automatically each time you open a Windows PowerShell prompt or the Windows PowerShell ISE.

- **Module logging:**
 - Logs events for running cmdlets in specific modules
 - Event ID 4103
- **Script block logging:**
 - Logs events for running script blocks, functions, and scripts
 - Suspicious script blocks are always logged
 - Event ID 4104
- **Transcription**
 - Records commands and results
 - Can be enabled by Group Policy in Windows PowerShell 5.0

Studyjní materiály Oskolení



Note: You require Windows PowerShell 5.0 to use **Start-Transcript** in the Windows PowerShell ISE.

In Windows PowerShell 5.0 and newer, you can enable transcription logging by using a GPO. The **Turn on PowerShell Transcription** setting is located in **\Policies\Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell**. You have the option to specify a directory for the transcript. If you do not specify a directory, then Windows PowerShell creates the transcript in the **Documents** folder. The **Include invocation headers** option adds a time stamp to the transcript that indicates when each command ran.



Note: If you specify a central folder on a computer for transcription, then you need to consider the file system security for that folder. You should configure the security permissions on the central folder to prevent users from monitoring each other's activities or deleting the files.

Question: If you run an external command at a Windows PowerShell prompt and get an unexpected error, how can you try to resolve it?

Lab: Practicing advanced techniques

Scenario

You are starting to become more proficient at using Windows PowerShell to accomplish administrative tasks for your organization. There are many tasks that require your immediate attention. To complete them, you decide to create scripts by using a wide variety of concepts.

Objectives

After completing this lab, you should be able to:

- Create a profile script.
- Verify the validity of an IP address.
- Report disk information.
- Query NTFS permissions.
- Create user accounts with passwords from a CSV file.

Lab Setup

Estimated Time: **120 minutes**

Virtual machines: **10961C-LON-DC1**, **10961C-LON-SVR1**, and **10961C-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

For this lab, you will use the available VM environment. Before beginning the lab, you must complete the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In Microsoft Hyper-V Manager, click **10961C-LON-DC1**, and then in the **Actions** pane, click **Start**.
3. In the **Actions** pane, click **Connect**. Wait until the VM starts.
4. Sign in by using the following credentials:
 - User name: **Administrator**
 - Password: **Pa55w.rd**
 - Domain: **Adatum**
5. Repeat steps 2 through 4 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Studijní materiály Okškolení

Exercise 1: Creating a profile script

Scenario

You perform many server administration tasks from a Windows PowerShell prompt. As part of your administration, you often need to place all servers and all domain controllers in a variable. To reduce your workload, you have decided to use a profile script to automatically populate **\$MyDCs** and **\$MyServers** each time a user opens the Windows PowerShell prompt.

The main task for this exercise is as follows:

1. Create a profile script.

► Task 1: Create a profile script

1. Identify a cmdlet that can query domain controllers.
2. Identify a computer object property that you can use with **Get-ADComputer** to query only servers.
3. Create a profile script that populates **\$MyDCs** and **\$MyServers**.
4. Place the script in the correct location to load only for your user account and only for the Windows PowerShell prompt.
5. Open a Windows PowerShell prompt and verify that the profile script populated the variables properly.
6. Open the Windows PowerShell ISE and verify that the profile script is not populating the variables.
7. Update the profile script so that the variables contain only computer names and not objects.

Results: After completing this exercise, you should have created a profile script.

Exercise 2: Verifying the validity of an IP address

Scenario

You have several scripts that obtain an IP address from user input. Some script users have been complaining that when they make a typing error, the script generates an error and stops, instead of allowing them to fix it. To improve your scripts, you are developing code that will verify the validity of an IP address based on its pattern and the values in each octet.

The main task for this exercise is as follows:

1. Verify the validity of an IP address.

► **Task 1: Verify the validity of an IP address**

1. Identify a regular expression that can verify the pattern of an IP address.
2. Identify a method for dividing the IP address into octets that you can evaluate to be between 0 and 255.
3. Create a script that requests an IP address from a user and then verifies the pattern and the value of each octet. The validity of the pattern and each octet should display on the screen.

Results: After completing this exercise, you should have created a script that verifies the validity of an IP address.

Exercise 3: Reporting disk information

Scenario

Over the past few months, there have been several instances where servers have experienced errors because of low amounts of free disk space. Your supervisor has asked you to create a script that queries disk space on a remote server and writes the information to the screen in aligned columns. The disk space should display in gigabytes (GB).

The main task for this exercise is as follows:

1. Report disk information.

► **Task 1: Report disk information**

1. Create a script that accepts the parameter `-ComputerName`.
2. Identify how to use **Get-Volume** to query information from a remote computer.
3. Filter the information from **Get-Volume** to include only volumes on hard drives.
4. Sort the volumes by drive letter to simplify reading.
5. Display the name of the queried computer on the screen.
6. Display a header for the columns **Drive**, **Size**, and **Free**. Choose an appropriate alignment for each column.
7. Display information for each volume, while using the same column alignment as the header. The **Size** and **Free** columns should be in GB and only include two decimal places.
8. Verify that the script is working.
9. Add a fourth column for **Status**. If the value in the **Free** column is less than 10 GB, show a status of **Low**.
10. Verify that the script is working.

Results: After completing this exercise, you should have created a script that reports disk space on a server.

Exercise 4: Querying NTFS permissions

Scenario

You would like to speed up the process of viewing NTFS permissions on files and folders from a Windows PowerShell prompt. To support this, you are creating a module that can show a summary of permissions or detailed NTFS permissions that **Get-Acl** retrieves.

The main task for this exercise is as follows:

1. Query NTFS permissions.

► Task 1: Query NTFS permissions

1. Create a script with a function named **Get-NTFS** that accepts a *-Path* parameter and a *-Full* parameter. The *-Full* parameter will indicate when detailed NTFS permissions should be displayed.
2. Identify how to display summary access rules from an ACL.
3. Identify how to display detailed access rules from an ACL.
4. Verify that you can query summary access rules and detailed access rules by using the function.
5. Add functionality to request a folder path when the user does not provide it as a parameter.
6. Convert the script to a module and copy it to the required location for a module.
7. Verify that you can call the function for summary and detailed access rules.

Results: After completing this exercise, you will have created a module that you can use to query NTFS permissions.

Exercise 5: Creating user accounts with passwords from a CSV file

Scenario

The Human Resources department has installed a new management system. When new employees are hired, the Human Resources department enters their details into the new system. Each day, the system generates a CSV file that contains an export of new employee information, which includes a temporary password that each new employee receives.

You decide to create a script that will create new user accounts from the information in the CSV file. The script should create each new user account in the organizational unit (OU) that matches their department. Additionally, each new user account should be enabled when the script is complete.

The main tasks for this exercise are as follows:

1. Create user accounts with a password from a CSV file.
2. Prepare for the end of the course.

► Task 1: Create user accounts with a password from a CSV file

1. Review the contents of **users.csv**.
2. Create a script that accepts the *-CsvFile* parameter.
3. In the script, verify that the user specifies a value for *-CsvFile*. If there is no value, then exit the script with an error message.

4. Import the CSV file, and then create user accounts based on the data in the CSV file. Configure the following options:
 - **Display name**
 - **UPN**
 - **Password**
 - **Organizational unit**
 - **Enable the account**
 - **Require the password to be changed next logon**
5. Verify that the script created the new user accounts in the correct OU.

Results: After completing this exercise, you will have created a script that will create new user accounts from a CSV file.

► Task 2: Prepare for the end of the course

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Question: When you created the value for the `-Path` parameter while creating user accounts, is there a way that you could have automatically identified the current domain?

Question: When you use the format operator to align columns, how do you know how wide to make the columns?

Module Review and Takeaways

Review Questions

Question: If you are using the format operator to display a number, how do you specify the number of decimal places to include?

Question: Your organization has decided to log all use of Windows PowerShell on domain controllers. How will you do this?

Course Evaluation

Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

Studijní materiály Okškolení

Studijní materiály Okškolení

Module 1: Getting started with Windows PowerShell

Lab A: Configuring Windows PowerShell

Exercise 1: Configuring the Windows PowerShell console application

- Task 1: Start the console application as Administrator and pin the Windows PowerShell icon to the taskbar

1. On **LON-CL1**, click **Start**.
2. Type **powersh** to display the Windows PowerShell icon.



Note: Make sure that the icon name displays **Windows PowerShell** and not **Windows PowerShell (x86)**.

3. Right-click **Windows PowerShell**, and then select **Run as administrator**.
4. Make sure that the window title bar reads **Administrator** and does not include the text **(x86)**. This indicates that it is the 64-bit console application and that an administrator is running it.
5. On the taskbar, right-click the **Windows PowerShell** icon, and then select **Pin to taskbar**.



Note: The Windows PowerShell console should now be open, run by **Administrator**, and available on the taskbar for future use.

- Task 2: Configure the Windows PowerShell console application

1. To configure Windows PowerShell to use the **Consolas** font:
 - a. Select the control box in the upper-left corner of the **Windows PowerShell** console window.
 - b. Select **Properties**.
 - c. In the “**Windows PowerShell**” **Properties** dialog box, select the **Font** tab, and then, in the **Font** list, select **Consolas**.
 - d. Select **16** in the **Size** list.
2. To select alternate display colors, on the **Colors** tab, review the available **Screen Text** and **Screen Background** colors.



Note: Experiment with various combinations. You can use the color picker to change colors quickly to improve readability.

3. To resize the window and remove the horizontal scroll bar:
 - a. On the **Layout** tab, in the **Window Size** settings, change the area’s **Width** and **Height** values until the **Windows PowerShell** console pane preview fits completely within the **Window Preview** area.
 - b. On the **Layout** tab, in the **Screen Buffer Size** settings, change the **Width** value to be the same as the **Width** value in the **Windows Size** settings.
4. Select **OK**. The console application should now be ready for use.

► **Task 3: Start a shell transcript**

- In the Windows PowerShell console, type the following command, and then press Enter:

```
Start-Transcript C:\DayOne.txt
```

 **Note:** You have now started a transcript of your Windows PowerShell session. It will save all the commands you type and also the command output to the text file until you run **Stop-Transcript** or close the Windows PowerShell window.

You can view the contents of the transcript at any time by opening **C:\DayOne.txt**.

Results: After completing this exercise, you will have opened and configured the Windows PowerShell console application and configured its appearance and layout.

Exercise 2: Configuring the Windows PowerShell ISE application

► **Task 1: Open the Windows PowerShell ISE application as Administrator**

1. In the Windows PowerShell console, type **ise**, and then press Enter.

 **Note:** This method of opening the ISE will work correctly only when an administrator is running the console.

2. Close the ISE window.
3. Right-click the **Windows PowerShell** icon on the taskbar and then select **Run ISE as Administrator**. You should now be running Windows PowerShell ISE as **Administrator**.

► **Task 2: Customize the appearance of the ISE to use the single-pane view, hide the Command pane, and adjust the font size**

1. To configure the ISE to use a single-pane view:
 - a. On the Windows PowerShell ISE toolbar, select the **Show Script Pane Maximized** option.
 - b. Select the **Hide Script Pane** up-arrow icon to display the console.
2. Select the **Show Command Add-on** option to view the **Command** pane, if it is not showing.
3. Select the **Show-Command Add-on** option to hide the **Command** pane.
4. Use the slider in the lower-right corner of the window to adjust the font size until you can read it comfortably.
5. Close the Windows PowerShell ISE and the Windows PowerShell windows.

Results: After completing this exercise, you will have customized the appearance of the Windows PowerShell Integrated Scripting Environment (ISE) application.

► **Task 3: Prepare for the next lab**

- Leave the virtual machines running for the next lab.

Lab B: Finding and running basic commands

Exercise 1: Finding commands

► Task 1: Find commands that will accomplish specified tasks

1. On **LON-CL1**, on the task bar, right-click **Windows PowerShell**, and then select **Run as Administrator**.
2. In the console, type one of the following commands, and then press Enter:

```
Get-Help *resolve*
```

or:

```
Get-Command *resolve*
```

or:

```
Get-Command -Verb resolve
```

 **Note:** The first two commands display a list of commands that use *Resolve* anywhere in their names. The third displays a list of commands that use the verb *Resolve* in their name. All three will return the same results in the lab environment. This should lead you to the **Resolve-DNSName** command.

3. In the console, type one of the following commands, and then press Enter:

```
Get-Help *adapter*
```

or:

```
Get-Command *adapter*
```

or:

```
Get-Command -Noun *adapter*
```

or:

```
Get-Command -Verb Set -Noun *adapter*
```

 **Note:** The first three commands display a list of commands that use *Adapter* in their names. The fourth displays a list of commands that have *Adapter* in their names and use the *Set* verb. This should lead you to the **Set-NetAdapter** command.

4. Run **Get-Help Set-NetAdapter** to view the help for that command. This should lead you to the **-MACAddress** parameter.

Studijní materiály Okškolení

5. In the console, type one of the following commands, and then press Enter:

```
Get-Help *sched*
```

or:

```
Get-Command *sched*
```

or:

```
Get-Module *sched* -ListAvailable
```

 **Note:** The first two commands display a list of commands that use *Sched* in their name. The third displays a list of modules with *Sched* in their name, which should lead you to the module **ScheduledTasks**. If you then run the command **Get-Command -Module *ScheduledTask***, you will see a list of commands in that module.

This should lead you to the **Enable-ScheduledTask** command.

6. In the console, type one of the following commands, and then press Enter:

```
Get-Command -Verb Block
```

or:

```
Get-Help *block*
```

 **Note:** These display a list of commands. This should lead you to the **Block-SmbShareAccess** command. Then, run **Get-Help Block-SmbShareAccess** to learn that the command applies a Deny entry to the file share discretionary access control list (DACL).

7. In the console, type the following command, and then press Enter:

```
Get-Help *branch*
```

 **Note:** This will cause the help system to conduct a full-text search, because no commands use *branch* in their names.

A list of topics containing the text *branch* displays, but none appear related to clearing the **BranchCache** cache.

8. In the console, type one of the following commands, and then press Enter:

```
Get-Help *cache*
```

or:

```
Get-Command *cache*
```

or:

```
Get-Command -Verb clear
```

 **Note:** The first two commands will display a list of commands containing *Cache* in the name. The third displays a list of commands using the verb *Clear* in the name. Either way, you should discover the **Clear-BCCache** command.

9. In the console, type one of the following commands, and then press Enter:

```
Get-Help *firewall*
```

or:

```
Get-Command *firewall*
```

or:

```
Get-Help *rule*
```

or:

```
Get-Command *rule*
```

 **Note:** These display a list of commands that use those words in their names. This should lead you to the **Get-NetFirewallRule** command.

10. In the console, type the following command, and then press Enter:

```
Get-Help Get-NetFirewallRule -Full
```

 **Note:** This will display the help for the command. This should let you discover the *-Enabled* parameter.

11. In the console, type the following command, and then press Enter:

```
Get-Help *address*
```

 **Note:** This will display a list of commands that use *address* in their names. This should lead you to the **Get-NetIPAddress** command.

12. In the console, type the following command, and then press Enter:

```
Get-Command -Verb suspend
```

 **Note:** This displays a list of commands that use the verb *Suspend* in their names. This should lead you to the **Suspend-PrintJob** command.

13. In the console, type one of the following commands, and then press Enter:

```
Get-Alias Type
```

or:

```
Get-Command -Noun *content*
```

 **Note:** The first command displays the alias definition for the **Type** command, which is the command used in **cmd.exe** to read text from a file. The second command returns a list of commands with *Content* in the noun portion of the name.

This should lead you to the **Get-Content** command.

Results: After completing this exercise, you will have demonstrated your ability to find new Windows PowerShell commands that perform specific tasks.

Exercise 2: Running commands

► Task 1: Run commands to accomplish specified tasks

1. Ensure you are signed in on the **LON-CL1** virtual machine as **Adatum\Administrator**.
2. To display a list of enabled firewall rules, in the console, type the following command, and then press Enter:

```
Get-NetFirewallRule -Enabled True
```

3. To display a list of all local IPv4 addresses, in the console, type the following command, and then press Enter:

```
Get-NetIPAddress -AddressFamily IPv4
```

4. To set the startup type of the Background Intelligent Transfer Service (BITS), in the console, type the following command, and then press Enter:

```
Set-Service -Name BITS -StartupType Automatic
```

5. To test the connection to **LON-DC1**, in the console, type the following command, and then press Enter:

```
Test-Connection -ComputerName LON-DC1 -Quiet
```



Note: This command returns only a True or False value, without any other output.

6. To display the newest 10 entries from the Security event log, in the console, type the following command, and then press Enter:

```
Get-EventLog -LogName Security -Newest 10
```

Results: After completing this exercise, you will have demonstrated your ability to run Windows PowerShell commands by using correct command-line syntax.

Exercise 3: Using About files

► Task 1: Locate and read About help files

1. Ensure you are still signed in to **LON-CL1** as **Adatum\Administrator** from the previous exercise.
2. To find operators used for wildcard string comparison, in the console, type the following command, and then press Enter:

```
Get-Help *comparison*
```

3. In the console, type the following command, and then press Enter:

```
Get-Help about_comparison_operators -ShowWindow
```

Notice the **-Like** operator in the **about_Comparison_Operators** help, which displays in a modal window.

4. To find the **-Like** operator, in the **Find** text box, type **wildcard**, and then select **Next**.
5. After reading the **about_Comparison_Operators** file, you should learn that typical operators are not case-sensitive. Specific case-sensitive operators are provided in **about_Comparison_Operators**.
6. To display the **COMPUTERNAME** environment variable, in the console, type the following command, and then press Enter:

```
$env:computername
```

You could read about this technique in **about_environment_variables**.

7. In the console, type the following command, and then press Enter:

```
Get-Help *signing*
```

8. In the console, type the following command, and then press Enter:

```
Get-Help about_signing
```

9. Read about code signing. You should learn that **makecert.exe** is used to create a self-signed digital certificate.

Results: After completing this exercise, you will have demonstrated your ability to locate help content in About files.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Module 2: Cmdlets for administration

Lab: Windows administration

Exercise 1: Creating and managing Active Directory objects

► **Task 1: Create a new organizational unit (OU) for a branch office**

1. On **LON-CL1**, click **Start** and then type **powershell**.
2. In the search results, right-click **Windows PowerShell**, and then click **Run as administrator**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
New-ADOrganizationalUnit -Name London
```

► **Task 2: Create group for branch office administrators**

- In the console, type the following command, and then press Enter:

```
New-ADGroup "London Admins" -GroupScope Global
```

► **Task 3: Create a user and computer account for the branch office**

1. In the console, type the following command, and then press Enter:

```
New-ADUser -Name Ty -DisplayName "Ty Carlson"
```

2. Type the following command, and then press Enter:

```
Add-ADGroupMember "London Admins" -Members Ty
```

3. Type the following command, and then press Enter:

```
New-ADComputer LON-CL2
```

► **Task 4: Move the group, user, and computer accounts to the branch office OU**

1. In the console, type the following command, and then press Enter:

```
Move-ADObject -Identity "CN=London Admins,CN=Users,DC=Adatum,DC=com" -TargetPath  
"OU=London,DC=Adatum,DC=com"
```

2. Type the following command, and then press Enter:

```
Move-ADObject -Identity "CN=Ty,CN=Users,DC=Adatum,DC=com" -TargetPath  
"OU=London,DC=Adatum,DC=com"
```

3. Type the following command, and then press Enter:

```
Move-ADObject -Identity "CN=LON-CL2,CN=Computers,DC=Adatum,DC=com" -TargetPath  
"OU=London,DC=Adatum,DC=com"
```

Results: After completing this exercise, you will have successfully identified and used commands for managing Active Directory objects in the Windows PowerShell command-line interface.

Exercise 2: Configuring network settings on Windows Server

► Task 1: Test the network connection and view the configuration

1. Switch to **LON-SVR1**.
2. Right-click the **Start** button, and then select **Windows PowerShell (Admin)**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Test-Connection LON-DC1
```

 **Note:** Note the speed of the connection so that you can compare it to the speed after you make changes.

4. In the console, type the following command, and then press Enter:

```
Get-NetIPConfiguration
```

 **Note:** Note the IP address, default gateway and Domain Name System (DNS) server.

► Task 2: Change the server IP address

1. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
New-NetIPAddress -InterfaceAlias Ethernet -IPAddress 172.16.0.15 -PrefixLength 16
```

2. In the console, type the following command and press Enter:

```
Remove-NetIPAddress -InterfaceAlias Ethernet -IPAddress 172.16.0.11
```

3. Type **Y** and press Enter for both confirmation prompts.

► Task 3: Change the DNS settings and default gateway for the server

1. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Set-DnsClientServerAddress -InterfaceAlias Ethernet -ServerAddress 172.16.0.12
```

2. In the console, type the following command, and then press Enter:

```
Remove-NetRoute -InterfaceAlias Ethernet -DestinationPrefix 0.0.0.0/0 -Confirm:$false
```

3. In the console window, type the following command, and then press Enter:

```
New-NetRoute -InterfaceAlias Ethernet -DestinationPrefix 0.0.0.0/0 -NextHop  
172.16.0.2
```

► **Task 4: Verify and test the changes**

- On **LON-SVR1**, in the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-NetIPConfiguration
```

 **Note:** Note the IP address, default gateway, and DNS server.

- 2. In the console, type the following command, and then press Enter:

```
Test-Connection LON-DC1
```

 **Note:** It now takes much longer to receive a response from **LON-DC1**.

Note that the actual time it takes may vary based on many factors. While the change should be large enough to be noticeable, it is possible you will not see much difference.

Results: After completing this exercise, you will have successfully identified and used Windows PowerShell commands for managing network configuration.

Exercise 3: Creating a website

► **Task 1: Install IIS on the server**

- On **LON-SVR1**, in the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Install-WindowsFeature Web-Server
```

 **Note:** Wait for Internet Information Services (IIS) to install.

► **Task 2: Create a folder on the server for the website files**

- On **LON-SVR1**, in the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
New-Item C:\inetpub\wwwroot\London -Type directory
```

► **Task 3: Create a new application pool for the website**

- In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
New-WebAppPool LondonAppPool
```

Studijní materiály Okškolení

► Task 4: Create the IIS website

1. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
New-WebSite London -PhysicalPath C:\inetpub\wwwroot\london -IPAddress 172.16.0.15 -ApplicationPool LondonAppPool
```

2. On the taskbar, click the **Internet Explorer** icon.
3. In the Address bar, type **172.16.0.15**, and then press Enter.

 **Note:** Internet Explorer displays an error message. The error message details give the physical path of the site, which should be **C:\inetpub\wwwroot\london**.

Results: After completing this exercise, you will have successfully identified and used Windows PowerShell commands that would be used as part of a standardized Web server configuration.

► Task 5: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-SVR1** and **10961C-LON-CL1**.

Module 3: Working with the Windows PowerShell pipeline

Lab A: Using the pipeline

Exercise 1: Selecting, sorting, and displaying data

► Task 1: Display the current day of the year

1. On **LON-CL1**, click **Start** and then type **powersh**.
2. In the search results, right-click **Windows PowerShell**, and then click **Run as administrator**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
help *date*
```



Note: Notice the **Get-Date** command.

4. In the console, type the following command, and then press Enter:

```
Get-Date | Get-Member
```



Note: Notice the **DayOfYear** property.

5. In the console, type the following command, and then press Enter:

```
Get-Date | Select-Object -Property DayOfYear
```

6. In the console, type the following command, and then press Enter:

```
Get-Date | Select-Object -Property DayOfYear | fl
```

► Task 2: Display information about installed hotfixes

1. In the console, type the following command, and then press Enter:

```
Get-Command *hotfix*
```



Note: Notice the **Get-Hotfix** command.

2. In the console, type the following command, and then press Enter:

```
Get-Hotfix | Get-Member
```



Note: The properties of the **Hotfix** object display. If needed, run **Get-Hotfix** to see some of the values that typically appear in those properties.

3. In the console, type the following command, and then press Enter:

```
Get-Hotfix | Select-Object -Property HotFixID, InstalledOn, InstalledBy
```

4. In the console, type the following command, and then press Enter:

```
Get-Hotfix | Select-Object -Property HotFixID, @{n='HotFixAge';e={(New-TimeSpan -Start $PSItem.InstalledOn).Days}}, InstalledBy
```

► Task 3: Display a list of available scopes from the DHCP server

1. In the console, type the following command, and then press Enter:

```
help *scope*
```

 **Note:** Notice the **Get-DHCPServerv4Scope** command.

2. In the console, type the following command, and then press Enter:

```
Help Get-DHCPServerv4Scope -ShowWindow
```

 **Note:** Notice the available parameters.

3. In the console, type the following command, and then press Enter:

```
Get-DHCPServerv4Scope -ComputerName LON-DC1
```

4. In the console, type the following command, and then press Enter:

```
Get-DHCPServerv4Scope -ComputerName LON-DC1 | Select-Object -Property ScopeId, SubnetMask, Name | fl
```

► Task 4: Display a sorted list of enabled Windows Firewall rules

1. In the console, type the following command, and then press Enter:

```
help *rule*
```

 **Note:** Notice the **Get-NetFirewallRule** command.

2. In the console, type the following command, and then press Enter:

```
Get-NetFirewallRule
```

3. In the console, type the following command, and then press Enter:

```
Help Get-NetFirewallRule -ShowWindow
```

4. In the console, type the following command, and then press Enter:

```
Get-NetFirewallRule -Enabled True
```

5. In the console, type the following command, and then press Enter:

```
Get-NetFirewallRule -Enabled True | Format-Table -wrap
```

6. In the console, type the following command, and then press Enter:

```
Get-NetFirewallRule -Enabled True | Select-Object -Property DisplayName,Profile,Direction,Action | Sort-Object -Property Profile, DisplayName | ft -GroupBy Profile
```

► Task 5: Display a sorted list of network neighbors

1. In the console, type the following command, and then press Enter:

```
help *neighbor*
```



Note: Notice the **Get-NetNeighbor** command.

2. In the console, type the following command, and then press Enter:

```
help Get-NetNeighbor -ShowWindow
```

3. In the console, type the following command, and then press Enter:

```
Get-NetNeighbor
```

4. In the console, type the following command, and then press Enter:

```
Get-NetNeighbor | Sort-Object -Property State
```

5. In the console, type the following command, and then press Enter:

```
Get-NetNeighbor | Sort-Object -Property State | Select-Object -Property IPAddress,State | Format-Wide -GroupBy State -AutoSize
```

► Task 6: Display information from the DNS name resolution cache

1. In the console, type the following command, and then press Enter:

```
Test-NetConnection LON-DC1
```

2. In the console, type the following command, and then press Enter:

```
Test-NetConnection LON-CL1
```

3. In the console, type the following command, and then press Enter:

```
help *cache*
```



Note: Notice the **Get-DnsClientCache** command.

4. In the console, type the following command, and then press Enter:

```
Get-DnsClientCache
```

Studijní materiály Okškolení

5. In the console, type the following command, and then press Enter:

```
Get-DnsClientCache | Select Name,Type,TimeToLive | Sort Name | Format-List
```



Note: Notice that the **Type** data does not return what you might expect—for example, A and CNAME. Instead, it returns raw numerical data. Each number maps directly to a record type, and you can filter for those types when you know the map: 1 = A, 5 = CNAME, and so on. Later in this module, you will learn how to add more filters to determine the numbers and their corresponding record types. You will notice a similar situation for other returned data, such as **Status** data.

6. Close all open windows.

Results: After completing this exercise, you should have produced several custom reports that contain management information from your environment.

► **Task 7: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Lab B: Filtering objects

Exercise 1: Filtering objects

► Task 1: Display a list of all the users in the Users container of Active Directory

1. On **LON-CL1**, click **Start** and then type **powersh**.
2. In the search results, right-click **Windows PowerShell**, and then click **Run as administrator**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
help *user*
```



Note: Notice the **Get-ADUser** command.

4. In the console, type the following command, and then press Enter:

```
Get-Help Get-ADUser -ShowWindow
```



Note: Notice that the **-Filter** parameter is mandatory. Review the examples for the command.

5. In the console, type the following command, and then press Enter:

```
Get-ADUser -Filter * | ft
```

6. In the console, type the following command, and then press Enter:

```
Get-ADUser -Filter * -SearchBase "cn=Users,dc=Adatum,dc=com" | ft
```

► Task 2: Create a report showing the Security event log entries that have the event ID 4624

1. In the console, type the following command, and then press Enter:

```
Get-EventLog -LogName Security |
Where EventID -eq 4624 | Measure-Object | fw
```

2. In the console, type the following command, and then press Enter:

```
Get-EventLog -LogName Security |
Where EventID -eq 4624 |
Select TimeWritten,EventID,Message
```

3. In the console, type the following command, and then press Enter:

```
Get-EventLog -LogName Security |
Where EventID -eq 4624 |
Select TimeWritten,EventID,Message -Last 10 | fl
```

Studijní materiály Okškolení

► **Task 3: Display a list of the encryption certificates installed on the computer**

1. In the console, type the following command, and then press Enter:

```
Get-ChildItem -Path CERT: -Recurse
```

2. To display the members of the objects, type the following command, and then press Enter:

```
Get-ChildItem -Path CERT: -Recurse |  
Get-Member
```

3. In the console, type one of the following commands, and then press Enter:

```
Get-ChildItem -Path CERT: -Recurse |  
Where HasPrivateKey -eq $False | Select-Object -Property FriendlyName,Issuer | fl
```

or:

```
Get-ChildItem -Path CERT: -Recurse |  
Where { $PSItem.HasPrivateKey -eq $False } | Select-Object -Property  
FriendlyName,Issuer | fl
```

4. In the console, type the following command, and then press Enter:

```
Get-ChildItem -Path CERT: -Recurse |  
Where { $PSItem.HasPrivateKey -eq $False -and $PSItem.NotAfter -gt (Get-Date) -and  
$PSItem.NotBefore -lt (Get-Date) } | Select-Object -Property NotBefore,NotAfter,  
FriendlyName,Issuer | ft -wrap
```

► **Task 4: Create a report that shows the disk volumes that are running low on space**

1. In the console, type the following command, and then press Enter:

```
Get-Volume
```

 **Note:** If you did not know the command name, you could have run **Help *volume*** to discover it.

2. In the console, type the following command, and then press Enter:

```
Get-Volume | Get-Member
```

 **Note:** Notice the **SizeRemaining** property.

3. In the console, type the following command, and then press Enter:

```
Get-Volume | Where-Object { $PSItem.SizeRemaining -gt 0 } | fl
```

4. In the console, type the following command, and then press Enter:

```
Get-Volume | Where-Object { $PSItem.SizeRemaining -gt 0 -and $PSItem.SizeRemaining /  
$PSItem.Size -lt .99 } | Select-Object DriveLetter, @{n='Size';e='{0:N2}' -f  
($PSItem.Size/1MB)}}
```

5. In the console, type the following command, and then press Enter:

```
Get-Volume | Where-Object { $PSItem.SizeRemaining -gt 0 -and $PSItem.SizeRemaining / $PSItem.Size -lt .1 }
```

 **Note:** This command might not produce any output on your lab computer if the computer has more than 10 percent free space on each of its volumes.

► **Task 5: Create a report that displays specified Control Panel items**

1. In the console, type the following command, and then press Enter:

```
help *control*
```

 **Note:** Notice the **Get-ControlPanelItem** command.

2. In the console, type the following command, and then press Enter:

```
Get-ControlPanelItem
```

3. In the console, type the following command, and then press Enter:

```
Get-ControlPanelItem -Category 'System and Security' | Sort Name
```

 **Note:** Notice that you do not have to use **Where-Object**.

4. In the console, type the following command, and then press Enter:

```
Get-ControlPanelItem -Category 'System and Security' | Where-Object -FilterScript { -not ($PSItem.Category -notlike '*System and Security*')} | Sort Name
```

Results: After completing this exercise, you should have used filtering to produce lists of management information that include only specified data and elements.

► **Task 6: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Studijní materiály Okškolení

Lab C: Enumerating objects

Exercise 1: Enumerating objects

► Task 1: Display a list of files on the E: drive of your computer

1. On **LON-CL1**, click **Start** and then type **powersh**.
2. In the search results, right-click **Windows PowerShell**, and then click **Run as administrator**.
3. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-ChildItem -Path E: -Recurse
```

4. In the console, type the following command, and then press Enter:

```
Get-ChildItem -Path E: -Recurse | Get-Member
```

 **Note:** Notice the **GetFiles** method in the list under **TypeName: System.IO.DirectoryInfo**.

 **Note:** You will learn why there are two lists of members for Get-ChildItem in Module 5, "Using PSProviders and PSDrives."

5. In the console, type the following command, and then press Enter:

```
Get-ChildItem -Path E: -Recurse | ForEach GetFiles
```

► Task 2: Use enumeration to produce 100 random numbers

1. In the console, type the following command, and then press Enter:

```
help *random*
```

 **Note:** Notice the **Get-Random** command.

2. In the console, type the following command, and then press Enter:

```
help Get-Random -ShowWindow
```

 **Note:** Notice the **-SetSeed** parameter.

3. In the console, type the following command, and then press Enter:

```
1..100
```

4. In the console, type the following command, and then press Enter:

```
1..100 |
ForEach { Get-Random -SetSeed $PSItem }
```

Studijní materiály Okškolení

► **Task 3: Run a method of a Windows Management Instrumentation (WMI) object**

1. Close all applications other than the **Windows PowerShell** console.
2. In the console, type the following command, and then press Enter:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges
```

3. In the console, type the following command, and then press Enter:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges |  
Get-Member
```



Note: Notice the **Reboot** method.



Note: The following command will reboot the machine you run it on.

4. In the console, type the following command, and then press Enter:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges |  
ForEach Reboot
```

Results: After completing this exercise, you should have written commands that manipulate multiple objects in the pipeline.

► **Task 4: Prepare for the next lab**

- At the end of this lab, keep the virtual machines running as they are needed for the next lab.

Lab D: Sending output to a file

Exercise 1: Converting objects

► Task 1: Update Active Directory user information

 **Note:** In this lab, long commands typically display on several lines. This helps to prevent unintended line breaks in the middle of commands. However, when you type these commands, you should type them as a single line. That line might wrap on your screen into multiple lines, but the command will still work. Press Enter only after typing the entire command.

1. Sign in to the **LON-CL1** as **Adatum\Administrator** with the password of **Pa55w.rd**.
2. On **LON-CL1**, click **Start** and then type **powersh**.
3. In the search results, right-click **Windows PowerShell**, and then click **Run as administrator**.
4. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City | Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} | Select-Object -Property Name,Department,City | Sort Name
```

5. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City | Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} | Set-ADUser -Office 'LON-A/1000'
```

6. In the **Administrator: Windows PowerShell** window, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City,Office | Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} | Select-Object -Property Name,Department,City,Office | Sort Name
```

► Task 2: Produce an HTML report listing the Active Directory users in the IT department

1. In the console, type the following command, and then press Enter:

```
help ConvertTo-Html -ShowWindow
```

2. In the console, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City,Office | Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} | Sort Name | Select-Object -Property Name,Department,City,Office | ConvertTo-Html -Property Name,Department,City -PreContent Users | Out-File E:\UserReport.html
```

3. To view the HTML file, type the following command, and then press Enter:

```
Invoke-Expression E:\UserReport.html
```

Studijní materiály Okškolení

4. In the console, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City,Office |  
Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} |  
Sort Name |  
Select-Object -Property Name,Department,City,Office |  
Export-Clixml E:\UserReport.xml
```

5. In Internet Explorer, in the address bar, type **E:\UserReport.xml**, and then press Enter.

6. In the console, type the following command, and then press Enter:

```
Get-ADUser -Filter * -Properties Department,City,Office |  
Where {$PSItem.Department -eq 'IT' -and $PSItem.City -eq 'London'} |  
Sort Name |  
Select-Object -Property Name,Department,City,Office |  
Export-Csv E:\UserReport.csv
```

7. In File Explorer, go to **E:**, right-click **UserReport.csv**, click **Open with**, and then click **NotePad**.

8. In File Explorer, go to **E:**, double-click **UserReport.csv**.

Results: After completing this exercise, you should have converted Active Directory user objects to different data formats.

► Task 3: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Studijní materiály Okškolení

Module 4: Understanding how the pipeline works

Lab: Working with pipeline parameter binding

Exercise: Predicting pipeline behavior

► Task 1: Review existing commands

 **Note:** For these tasks, you may run individual commands and **Get-Member** to see what kinds of objects the commands produce. You may also view the Help for any of these commands. However, do not run the whole command as shown. If you do run the whole command, it might produce an error. The error does not mean the command is written incorrectly.

1. This command's intent is to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Get-Service -Name *
```

Will the command achieve the goal?

No. **Get-Service** does not accept **ADComputer** objects from the pipeline.

2. This command's intent is to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-Service -Name *
```

Will the command achieve the goal?

Yes. The **-ComputerName** parameter accepts pipeline input by using **ByPropertyName**.

3. This command's intent is to query an object from every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-WmiObject -Class Win32_BIOS
```

Will the command achieve the goal?

No. The **-ComputerName** parameter of **Get-WmiObject** does not accept pipeline input.

4. This command's intent is to list the services that are running on every computer in the domain:

```
Get-Service -ComputerName (Get-ADComputer -Filter *)
```

Will the command achieve the goal?

No. The **-ComputerName** parameter cannot accept objects of the type **ADComputer**.

5. This command's intent is to list the Security event log entries from the server **LON-DC1**:

```
Get-EventLog -LogName Security -ComputerName (Get-ADComputer -LON-DC1 | Select -ExpandProperty Name)
```

Will the command achieve the goal?

Yes. The parenthetical command produces objects of the type **String** that are computer names. The **-ComputerName** parameter accepts those.

Studijní materiály Okškolení

► Task 2: Write new commands that perform the specified tasks

 **Note:** In each of these tasks, you are to write a command that achieves a specified goal. *Do not run these commands.* Write them on paper.

You may run individual commands and pipe their output to **Get-Member** to see what objects those commands produce. You may also read the Help for any command.

1. Write a command that uses **Get-EventLog** to display the most recent 50 System event log entries from each computer in the domain.

Because **Get-EventLog** does not accept pipeline input for its **-ComputerName** parameter, you must use the following command:

```
Get-EventLog -LogName System -Newest 50 -ComputerName (Get-ADComputer -Filter * | Select-Object -ExpandProperty Name)
```

2. Write a command that uses **Set-Service** to set the start type of the **WinRM** service to **Auto** on every computer in the domain. Do not use a parenthetical command.

Because the **-ComputerName** parameter of **Set-Service** accepts pipeline input by using **ByPropertyName**, you can write the following command:

```
Get-ADComputer -Filter * | Select-Object @{n='ComputerName';e={$PSItem.Name}} | Set-Service -Name WinRM -StartupType Auto
```

You must use **Select-Object** because the **ADComputer** objects have a **Name** property, not a **ComputerName** property.

Results: After completing this exercise, you will have successfully reviewed and written several commands in the Windows PowerShell command-line interface.

► Task 3: Prepare for the next module

When you finish the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Module 5: Using PSProviders and PSDrives

Lab: Using PSProviders and PSDrives

Exercise 1: Creating files and folders on a remote computer

► Task 1: Create a new folder on a remote computer

1. On **LON-CL1**, click **Start** and then type **powershell**.
2. In the results list, right-click **Windows PowerShell** and then click **Run as administrator**.
3. In the **Administrator: Windows PowerShell** console, type the following command, and then press Enter:

```
Get-Help New-Item -ShowWindow
```

Notice the **-Name** and **-ItemType** parameters, and then review the example commands.

4. In the console, type the following command, and then press Enter:

```
New-Item -Path \\Lon-Svr1\C$\ -Name ScriptShare -ItemType Directory
```

► Task 2: Create a new PSDrive mapping to the remote file folder

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
Get-Help New-PSDrive -ShowWindow
```

Notice the **-Name**, **-Root**, and **-PSProvider** parameters. Review the example commands.

2. In the console, type the following command, and then press Enter:

```
New-PSDrive -Name ScriptShare -Root \\Lon-Svr1\c$\ScriptShare -PSProvider FileSystem
```

► Task 3: Create a file on the mapped drive

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
Get-Help Set-Location -ShowWindow
```

2. In the console, type the following command, and then press Enter:

```
Set-Location ScriptShare:
```

3. In the console, type the following command, and then press Enter:

```
New-Item script.txt
```

4. In the console, type the following command, and then press Enter:

```
Get-ChildItem
```

Verify that the **script.txt** file is listed.

Results: After completing this exercise, you should have successfully created a new folder and file on a remote computer and mapped a drive to that folder.

Exercise 2: Creating a registry key for your future scripts

► Task 1: Create the registry key to store script configurations

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
Get-ChildItem -Path HKCU:\Software
```

2. In the console, type the following command, and then press Enter:

```
New-Item -Path HKCU:\Software -Name Scripts
```

► Task 2: Create a new registry setting to store the name of the PSDrive

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
Set-Location HKCU:\Software\Scripts
```

2. In the console, type the following command, and then press Enter:

```
New-ItemProperty -Path HKCU:\Software\Scripts -Name "PSDriveName" -Value  
"ScriptShare"
```

3. In the console, type the following command, and then press Enter:

```
Get-ItemProperty . -Name PSDriveName
```

Results: After completing this exercise, you should have successfully created a registry key to store configuration data and added a registry setting that stores the name of a **PSDrive**. You also should have verified that you can retrieve the value of that setting by using Windows PowerShell.

Exercise 3: Creating a new Active Directory group

► Task 1: Create a PSDrive that maps to the Users container in AD DS

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
Import-Module ActiveDirectory
```

2. In the console, type the following command, and then press Enter:

```
New-PSDrive -Name "AdatumUsers" -Root "CN=users,DC=Adatum,DC=com" -PSProvider  
ActiveDirectory
```

3. In the console, type the following command, and then press Enter:

```
Set-Location "AdatumUsers:"
```

► Task 2: Create the London Developers group

1. In the **Windows PowerShell** console, type the following command, and then press Enter:

```
New-Item -ItemType group -Path . -Name "CN=London Developers"
```

2. In the console, type the following command, and then press Enter:

```
Get-ChildItem .
```

Verify that the new group is listed.

Results: After completing this exercise, you should have successfully created an Active Directory group by using the **ActiveDirectory** provider.

► Task 3: Prepare for the next module

When you finish the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Studijní materiály Okškolení

Module 6: Querying management information by using CIM and WMI

Lab: Working with CIM and WMI

Exercise 1: Querying information by using WMI

► Task 1: Query IP addresses

1. On **LON-CL1**, on the taskbar, in the **Ask me anything** box, type **PowerShell**. In the list that is returned, right-click **Windows PowerShell**, and then click **Run as administrator**.
2. To find a repository class that lists the IP addresses being used by a computer, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*configuration*' |
Sort Name
```

Notice the **Win32_NetworkAdapterConfiguration** class.

3. To retrieve all instances of the class showing DHCP IP addresses, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration |
Where DHCPEnabled -eq $False |
Select IPAddress
```

Remember that you can run the first command and pipe its output to **Get-Member** to see the properties that are available.

► Task 2: Query operating-system version information

1. To find a repository class that lists operating-system information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*operating*' |
Sort Name
```

Notice the **Win32_OperatingSystem** class.

2. To display a list of properties for the class, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_OperatingSystem | Get-Member
```

3. Notice the **Version**, **ServicePackMajorVersion**, and **BuildNumber** properties.
4. To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_OperatingSystem |
Select Version,ServicePackMajorVersion,BuildNumber
```

► Task 3: Query computer-system hardware information

- To find a repository class that displays computer-system information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*system*' |
Sort Name
```

Notice the **Win32_ComputerSystem** class.

- To display a list of instance properties and values, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -class Win32_ComputerSystem |
Format-List -Property *
```

Remember that **Get-Member** does not display property values, but **Format-List** can.

- To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -class Win32_ComputerSystem |
Select Manufacturer,Model,@{n='RAM';e={$PSItem.TotalPhysicalMemory}}
```

► Task 4: Query service information

- To find a repository class that contains information about services, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*service*' |
Sort Name
```

Notice the **Win32_Service** class.

- To display a list of instance properties and values, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_Service |
FL *
```

- To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_Service -Filter "Name LIKE '%S%'" |
Select Name,State,StartName
```

- Leave the **Windows PowerShell** console open for the next exercise.

Results: After completing this exercise, you should have queried repository classes by using WMI commands.

Exercise 2: Querying information by using CIM

► Task 1: Query user accounts

- To find a repository class that lists user accounts, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*user*' |
Sort Name
```

Notice the **Win32_UserAccount** class.

- To display a list of class properties, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -Class Win32_UserAccount |
Get-Member
```

- To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -Class Win32_UserAccount |
Format-Table -Property Caption,Domain,SID,FullName,Name
```

Notice the returned list of all domain and local accounts.

► Task 2: Query BIOS information

- To find a repository class that contains BIOS information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*bios*' |
Sort Name
```

Notice the **Win32_BIOS** class.

- To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -Class Win32_BIOS
```

► Task 3: Query network adapter configuration information

- To display a list of all the local **Win32_NetworkAdapterConfiguration** instances, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -Classname Win32_NetworkAdapterConfiguration
```

- To display a list of all the **Win32_NetworkAdapterConfiguration** instances that exist on **LON-DC1**, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -Classname Win32_NetworkAdapterConfiguration -ComputerName LON-DC1
```

Studijní materiály pro školení

► Task 4: Query user group information

- To find a repository class that lists user groups, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -namespace root\cimv2 -list |
Where Name -like '*group*' |
Sort Name
```

Notice the **Win32_Group** class.

- To display the specified information, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-CimInstance -ClassName Win32_Group -ComputerName LON-DC1
```

- Leave the **Windows PowerShell** console open for the next exercise.

Results: After completing this exercise, you should have queried repository classes by using CIM commands.

Exercise 3: Invoking methods

► Task 1: Invoke a CIM method

- To restart **LON-DC1**, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Invoke-CimMethod -ClassName Win32_OperatingSystem -ComputerName LON-DC1 -MethodName Reboot
```

Notice the response that shows **ReturnValue=0** and **PSCoputerName =LON-DC1**.

- Switch to the **LON-DC1** virtual machine, and see it restarting.

► Task 2: Invoke a WMI method

- Switch back to the **LON-CL1** virtual machine.
- Right-click the **Start** button, and then click **Computer Management**.
- Expand **Services and Applications**, and then click **Services**.
- Locate the **Windows Remote Management** (WS-Management) service, and then note the **Startup Type**. (The service is also known by the WinRM abbreviation, which is called out in the service description).
- To change the start mode of the specified service, type the following command in the **Windows PowerShell** console, and then press Enter:

```
Get-WmiObject -Class Win32_Service -Filter "Name='WinRM'" |
Invoke-WmiMethod -Name ChangeStartMode -Argument 'Automatic'
```

6. Verify that the status of the WinRM service has changed.

Results: After completing this exercise, you should have used CIM and WMI commands to invoke methods of repository objects.

► **Task 3: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1**.

Studijní materiály Okškolení

Module 7: Working with variables, arrays, and hash tables

Lab: Working with variables

Exercise 1: Working with variable types

► Task 1: Use string variables

1. On **LON-CL1**, click the **Start** button, and then type **powersh**.
2. In the results list, right-click **Windows PowerShell** and then click **Run as administrator**.
3. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$logPath = "C:\Logs\"
```

4. Type the following command, and then press Enter:

```
$logPath.GetType()
```

5. Type the following command, and then press Enter:

```
$logPath | Get-Member
```

6. Type the following command, and then press Enter:

```
$logFile = "log.txt"
```

7. Type the following command, and then press Enter:

```
$logPath += $logFile
```

8. Type the following command, and then press Enter:

```
$logPath
```

9. Type the following command, and then press Enter:

```
$logPath.Replace("C:", "D:")
```

10. Type the following command, and then press Enter:

```
$logPath = $logPath.Replace("C:", "D:")
```

11. Type the following command, and then press Enter:

```
$logPath
```

12. Leave the Windows PowerShell prompt open for the next task.

► Task 2: Use DateTime variables

1. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$today = Get-Date
```

2. Type the following command, and then press Enter:

```
$today.GetType()
```

Studijní materiály Okškolení

3. Type the following command, and then press Enter:

```
$today | Get-Member
```

4. Type the following command, and then press Enter:

```
$logFile = [string]$today.Year + "-" + $today.Month + "-" + $today.Day + "-" +  
$today.Hour + "-" + $today.Minute + ".txt"
```

5. Type the following command, and then press Enter:

```
$cutOffDate = $today.AddDays(-30)
```

6. Type the following command, and then press Enter:

```
Get-ADUser -Properties LastLogonDate -Filter {LastLogonDate -gt $cutOffDate}
```

7. Leave the Windows PowerShell prompt open for the next exercise.

Results: After completing this exercise, you should have manipulated multiple types of variables.

Exercise 2: Using arrays

► Task 1: Use an array to update the department for users

1. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$mktgUsers = Get-ADUser -Filter {Department -eq "Marketing"} -Properties Department
```

2. Type the following command, and then press Enter:

```
$mktgUsers.Count
```

3. Type the following command, and then press Enter:

```
$mktgUsers[0]
```

4. Type the following command, and then press Enter:

```
$mktgUsers | Set-ADUser -Department "Business Development"
```

5. Type the following command, and then press Enter:

```
$mktgUsers | Format-Table Name,Department
```

6. Review the output and verify that the Department values in the \$mktgUsers variable have not changed.

7. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ADUser -Filter {Department -eq "Marketing"}
```

- Type the following command, and then press Enter:

```
Get-ADUser -Filter {Department -eq "Business Development"}
```

- Leave the Windows PowerShell prompt open for the next task.

► Task 2: Use an arraylist

- At the Windows PowerShell prompt, type the following command, and then press Enter:

```
[System.Collections.ArrayList]$computers="LON-SRV1","LON-SRV2","LON-DC1"
```

- Type the following command, and then press Enter:

```
$computersFixedSize
```

- Type the following command, and then press Enter:

```
$computers.Add("LON-DC2")
```

- Type the following command, and then press Enter:

```
$computers.Remove("LON-SRV2")
```

- Type the following command, and then press Enter:

```
$computers
```

- Leave the Windows PowerShell prompt open for the next exercise.

Results: After completing this exercise, you should have manipulated arrays and arraylists.

Exercise 3: Using hash tables

► Task 1: Use a hash table

- At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$mailList=@{"Frank"="Frank@fabriakm.com";"Libby"="LHayward@contoso.com";"Matej"="MSTaojanov@tailspintoyso.com"}
```

- Type the following command, and then press Enter:

```
$mailList
```

- Type the following command, and then press Enter:

```
$mailList.Libby
```

- Type the following command, and then press Enter:

```
$mailList.Libby="Libby.Hayward@contoso.com"
```

- Type the following command, and then press Enter:

```
$mailList.Add("Stela","Stela.Sahiti")
```

Studyjní materiály Okškolení

6. Type the following command, and then press Enter:

```
$mailList.Remove("Frank")
```

7. Type the following command, and then press Enter:

```
$mailList
```

8. Close the Windows PowerShell prompt.

Results: After completing this exercise, you should have manipulated a hash table.

► Task 2: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Module 8: Basic scripting

Lab: Basic scripting

Exercise 1: Signing a script

► Task 1: Install a code signing certificate

1. On **LON-CL1**, click **Start**, type **mmc**, and click **mmc**.
2. In the **MMC** console, click **File**, and then click **Add/Remove Snap-in**.
3. In the **Add or Remove Snap-ins** window, click **Certificates**, and then click **Add**.
4. In the **Certificates snap-in** dialog box, click **My user account**, and then click **Finish**.
5. In the **Add or Remove Snap-ins** window, click **OK**.
6. In the **MMC** console, expand **Certificates - Current User**, and then click **Personal**.
7. Right-click **Personal**, point to **All Tasks**, and then click **Request New Certificate**.
8. In the **Certificate Enrollment** wizard, on the **Before You Begin** page, click **Next**.
9. On the **Select Certificate Enrollment Policy** page, click **Active Directory Enrollment Policy**, and then click **Next**.
10. On the **Request Certificates** page, select the **Adatum Code Signing** check box, and then click **Enroll**.
11. On the **Certificate Installation Results** page, click **Finish**.
12. In the **MMC** console, expand **Personal**, and then click **Certificates** to verify that the new code signing certificate is present.
13. Close the **MMC** console and click **No** at the prompt to save the console settings.

► Task 2: Digitally sign a certificate

1. Click the **Start** button, type **Powersh**, and then click **Windows PowerShell**.
2. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
```

3. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
$cert = Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
```

4. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod08\Labfiles
```

5. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Rename-Item HelloWorld.txt HelloWorld.ps1
```

6. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-AuthenticodeSignature -FilePath HelloWorld.ps1 -Certificate $cert
```

Studijní materiály Okškolení

► **Task 3: Set the execution policy**

1. At the Windows PowerShell prompt, type the following command, and then press Enter. Type **Y** at the prompt and press Enter:

```
Set-ExecutionPolicy AllSigned
```

2. At the Windows PowerShell prompt, type the following command, and then press Enter. You may be asked if you want to run software from the untrusted publisher. Type **A** and then press Enter:

```
.\HelloWorld.ps1
```

3. At the Windows PowerShell prompt, type the following command, and then press Enter. Type **Y** at the prompt and press Enter:

```
Set-ExecutionPolicy Unrestricted
```

4. Close the Windows PowerShell prompt.

Results: After completing this exercise, you should have digitally signed a script and modified the script execution policy.

Exercise 2: Processing an array with a ForEach loop

► **Task 1: Create a test group**

1. On **LON-CL1**, click **Start**, type **Powershell**, and then click **Windows PowerShell**.
2. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
New-ADGroup -Name IPPhoneTest -GroupScope Universal -GroupCategory Security
```

3. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Move-ADObject "CN=IPPhoneTest,CN=Users,DC=Adatum,DC=com" -TargetPath  
"OU=IT,DC=Adatum,DC=com"
```

4. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Add-ADGroupMember IPPhoneTest -Members Abbi,Ida,Parsa,Tonia
```

► **Task 2: Create a script to configure the ipPhone attribute**

- A script that performs these tasks is located at **E:\Mod08\Labfiles\10961C_Mod08_Ex2_LAK.txt**.

Results: After completing this exercise, you should have configured an Active Directory attribute by using a **ForEach** loop.

Exercise 3: Processing items by using If statements

► **Task 1: Create services.txt with service names**

1. Click **Start**, type **powersh**, and then click **Windows PowerShell**.
2. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Set-Location E:\Mod08\Labfiles
```

3. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
New-Item services.txt -ItemType File
```

4. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Service "Print Spooler" | Select -ExpandProperty Name | Out-File services.txt - Append
```

5. At the Windows PowerShell prompt, type the following command, and then press Enter:

```
Get-Service "Windows Time" | Select -ExpandProperty Name | Out-File services.txt - Append
```

► **Task 2: Create a script that starts stopped services**

- A script that performs these tasks is located at **E:\Mod08\Labfiles\10961C_Mod08_Ex3_LAK.txt**.

Results: After completing this exercise, you should have created a script that starts a list of services that are defined in a text file.

Exercise 4: Creating a random password

► **Task 1: Create a script that generates random passwords**

- A script that performs these tasks is located at **E:\Mod08\Labfiles\10961C_Mod08_Ex4_LAK.txt**.

Results: After completing this exercise, you should have created a script that generates a random password.

Exercise 5: Creating users based on a CSV file

► **Task 1: Create AD DS users from a CSV file**

- A script that performs these tasks is located at **E:\Mod08\Labfiles\10961C_Mod08_Ex5_LAK.txt**.

Results: After completing this exercise, you should have created users based on the data in a CSV file.

► **Task 2: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Module 9: Advanced scripting

Lab A: Accepting data from users

Exercise 1: Querying disk information from remote computers

- ▶ Task 1: Create a script that queries disk information with current credentials
- Perform all tasks using **LON-CLI**. A script that performs these tasks is located at:
E:\Mod09\Labfiles\10961C_Mod09_LabA_Ex1_LAK.txt.

Results: After completing this exercise, you will have created a script to query disk information from remote computers.

Exercise 2: Updating the script to use alternate credentials

- ▶ Task 1: Update the script to use alternate credentials
- A script that performs these tasks is located at:
E:\Mod09\Labfiles\10961C_Mod09_LabA_Ex2_LAK.txt.

Results: After completing this exercise, you will have updated the script to accept alternate credentials.

Exercise 3: Documenting a script

- ▶ Task 1: Document a script
- A script that performs these tasks is located at:
E:\Mod09\Labfiles\10961C_Mod09_LabA_Ex3_LAK.txt.
- ▶ Task 2: Prepare for the next lab
- Keep the virtual machines running as you will need them for the next lab.

Results: After completing this exercise, you will have documented a script.

Lab B: Implementing functions and modules

Exercise 1: Creating a logging function

► Task 1: Create a logging function

- A script that performs these tasks is located at:
E:\Mod09\Labfiles\10961C_Mod09_LabB_Ex1_LAK.txt.

Results: After completing this exercise, you should have created a logging function.

Exercise 2: Adding error handling to a script

► Task 1: Add error handling to a script

- A script that performs these tasks is located at:
E:\Mod09\Labfiles\10961C_Mod09_LabB_Ex2_LAK.txt.

Results: After completing this exercise, you will have added error handling to a script.

Exercise 3: Converting a function to a module

► Task 1: Convert a function to a module

1. In Windows PowerShell ISE, remove the line that calls the function. Only the function should be in the file.
2. Click the **File** menu and click **Save As**.
3. In the **Save As** dialog box, in the address bar, type **C:\Program Files\WindowsPowerShell\Modules** and press Enter.
4. Click **New folder**, type **LogFunction**, and press Enter.
5. Double-click **LogFunction**.
6. In the **File name** box, type **LogFunction.psm1** and then click **Save**.
7. At a Windows PowerShell prompt, type **Write-L** and press Tab. Successfully using Tab completion verifies that the function is loaded.

► **Task 2: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Results: After completing this exercise, you will have converted a function to a module.

Studijní materiály Okškolení

Module 10: Administering remote computers

Lab A: Using basic remoting

Exercise 1: Enabling remoting on the local computer

► Task: Enable remoting for incoming connections

1. Ensure that you are signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. Click the **Start** menu, and then type **powershell**.
3. In the results list, right-click **Windows PowerShell**, and click **Run as administrator**.
4. To ensure you have the correct execution policy in place, in the Windows PowerShell command window, type the following command, and then press Enter:

```
Set-ExecutionPolicy RemoteSigned
```

5. Click **Yes** or type **Y** to confirm the change.
6. On the **LON-CL1** computer, run the following command:

```
Enable-PSremoting
```

Answer Yes to all prompts by pressing **Y**. This will enable remoting.

7. To find a command that can list session configurations, type the following command, and then press Enter:

```
help *sessionconfiguration*
```

Notice the **Get-PSSessionConfiguration** command.

8. To list session configurations, type the following command, and then press Enter:

```
Get-PSSessionConfiguration
```

9. Verify that two to four session configurations were created.

Results: After completing this exercise, you should have enabled remoting on the client computer.

Exercise 2: Performing one-to-one remoting

► Task 1: Connect to the remote computer and install an operating system feature on it

1. Ensure that you are still signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. On **LON-CL1**, to establish a one-to-one connection to **LON-DC1**, type the following command in Windows PowerShell, and then press Enter:

```
Enter-PSSession -ComputerName LON-DC1
```

3. After you are connected, to install the Network Load Balancing (NLB) feature on **LON-DC1**, type the following command, and then press Enter:

```
Install-WindowsFeature NLB
```

4. Wait for the command to complete.
5. To disconnect, type the following command, and then press Enter:

```
Exit-PSSession
```

► Task 2: Test multi-hop remoting

1. To establish a one-to-one remoting connection to **LON-DC1**, type the following command, and then press Enter:

```
Enter-PSSession -ComputerName LON-DC1
```

2. To establish a connection from **LON-DC1** to **LON-CL1**, type the following command, and then press Enter:

```
Enter-PSSession -ComputerName LON-CL1
```

You should receive an error that is indicative of the second hop. By default, you cannot establish a connection through an already-established connection.

3. To close the connection, type the following command, and then press Enter:

```
Exit-PSSession
```

► Task 3: Observe remoting limitations

1. Ensure that you are signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. To establish a one-to-one connection to **LON-CL1**, type the following command, and then press Enter:

```
Enter-PSSession -ComputerName localhost
```

3. Type the following command, and then press Enter:

```
Notepad
```

Notice that the shell seems to stop responding while it waits for Notepad to open, because Notepad is a graphical application, and the shell has no way to display the graphical user interface (GUI).

4. Press Ctrl+C to cancel the process and return to a shell prompt.
5. To disconnect, type the following command, and then press Enter:

```
Exit-PSSession
```

Results: After completing this exercise, you should have connected to a remote computer, and performed maintenance tasks on it.

Exercise 3: Performing one-to-many remoting

► **Task 1: Retrieve a list of physical network adapters from two computers**

1. Ensure that you are still signed into the **10961C-LON-CL1** virtual machine as **Adatum\Administrator** with the password **Pa55w.rd**.
2. On **LON-CL1**, to find a command that can list network adapters, type the following command, and then press Enter:

```
help *adapter*
```

Note the **Get-NetAdapter** command.

3. To view the Help for the command, type the following command, and then press Enter:

```
help Get-NetAdapter
```

Note the **-Physical** parameter.

4. To run the command on **LON-DC1** and **LON-CL1** by means of remoting, type the following command, and then press Enter:

```
Invoke-Command -ComputerName LON-CL1,LON-DC1 -ScriptBlock { Get-NetAdapter -Physical }
```

► **Task 2: Compare the output of a local command to that of a remote command**

1. To view the members of a **Process** object, type the following command, and then press Enter:

```
Get-Process | Get-Member
```

2. To view the members from a remote **Process** object, type the following command, and then press Enter:

```
Invoke-Command -ComputerName LON-DC1 -ScriptBlock { Get-Process } | Get-Member
```

Note that the second set of results only includes two **MemberType** of **Methods; GetType** and **Tostring**. This is because the remote value **TypeName** is deserialized in comparison to the local output.

► **Task 3: Prepare for the next lab**

- Keep the virtual machines running for the next lab. Do not revert the virtual machines.

Results: After completing this exercise, you will have run commands against multiple remote computers.

Studyjní materiály Okškolení

Lab B: Using PSSessions

Exercise 1: Using implicit remoting

► Task 1: Create a persistent remoting connection to a server

1. On **10961C-LON-CL1**, sign in as **Adatum\Administrator** with the password **Pa55w.rd**.
2. Click the **Start** menu, and then type **powersh**.
3. In the results list, right-click **Windows PowerShell**, and click **Run as administrator**.
4. In the Windows PowerShell command window, create a persistent connection to **LON-DC1** and store it in a variable. Type the following command, and then press Enter:

```
$dc = New-PSSession -ComputerName LON-DC1
```

5. To view the PSSession list in the variable, type the following command, and then press Enter:

```
$dc
```

Verify that the connection is available.

► Task 2: Import and use a module from a server

1. To display a list of modules on **LON-DC1**, type the following command, and then press Enter:

```
Get-Module -ListAvailable -PSSession $dc
```

2. To find a module on **LON-DC1** that can work with Server Message Block (SMB) shares, type the following command, and then press Enter:

```
Get-Module -ListAvailable -PSSession $dc |  
Where { $_.Name -Like '*share*' }
```

3. To import the module from **LON-DC1** to your local computer, and to add the prefix **DC** to the important commands' nouns, type the following command, and then press Enter:

```
Import-Module -PSSession $dc -Name SMBShare -Prefix DC
```

4. To display a list of shares on **LON-DC1**, type the following command, and then press Enter:

```
Get-DCSMBShare
```

Because this command implicitly runs on **LON-DC1**, the command will display shares on that computer.

5. To display a list of shares on the local computer, type the following command, and then press Enter:

```
Get-SMBShare
```

Because you added the **DC** prefix to the imported commands, the local commands are still available by their original name.

Studijní materiály OKškolení

► **Task 3: Close all open remoting connections**

1. In the Windows PowerShell command window, type the following command, and then press Enter:

```
Get-PSSession | Remove-PSSession
```

2. The command to verify that the remoting session has been closed is not explicitly called out in the sample answer script provided at E:\Mod10\Labfiles\ImplicitRemoting.ps1.txt. To verify the remoting connection has been closed, type the following command, and then press Enter:

```
Get-PSSession
```

Verify no sessions are returned.

Results: After completing this exercise, you should have used implicit remoting to import and run commands from a remote computer.

Exercise 2: Managing multiple computers

► **Task 1: Create PSSessions to two computers**

1. Ensure that you are still signed in to **10961C-LON-CL1** as **Adatum\Administrator** with the password **Pa55w.rd**.
2. Open Windows PowerShell.
3. To create PSSessions to **LON-CL1** and **LON-DC1**, and to save those in a variable, type the following command, and then Enter:

```
$computers = New-PSSession -ComputerName LON-CL1,LON-DC1
```

4. To verify the connections, type the following command, and then press Enter:

```
$computers
```

Verify that two connections display as available.

► **Task 2: Create a report that displays Windows Firewall rules from two computers**

1. To find a module capable of working with network security, type the following command, and then press Enter:

```
Get-Module *security* -ListAvailable
```

Note the NetSecurity module.

2. To load the module into memory on **LON-CL1** and **LON-DC1**, type the following command, and then press Enter:

```
Invoke-Command -Session $computers -ScriptBlock { Import-Module NetSecurity }
```

Studijní materiály Okškolení

3. To find a command that can display Windows Firewall rules, type the following command, and then press Enter:

```
Get-Command -Module NetSecurity
```

Note the **Get-NetFirewallRule** command. If you would like to review the Help for the command, type the following command, and then press Enter:

```
Help Get-NetFirewallRule -ShowWindow
```

-  **Note:** If Help is not displaying correctly, run the commands from steps 1 to 3 in the Windows PowerShell command window as administrator rather than in Windows PowerShell ISE.

4. To display a list of enabled firewall rules on **LON-DC1** and **LON-CL1**, type the following command, and then press Enter:

```
Invoke-Command -Session $computers -ScriptBlock { Get-NetFirewallRule -Enabled True }  
|  
Select Name,PSComputerName
```

5. To unload the module on **LON-DC1** and **LON-CL1**, type the following command, and then press Enter:

```
Invoke-Command -Session $computers -ScriptBlock { Remove-Module NetSecurity }
```

► Task 3: Create and display an HTML report that displays local disk information from two computers

1. To display a list of local hard drives filtered to include only those with a drive type of 3, type the following command, and then press Enter:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3"
```

2. To run the same command on **LON-DC1** and **LON-CL1** by means of remoting, type the following command, and then press Enter:

```
Invoke-Command -Session $computers -ScriptBlock { Get-WmiObject -Class  
Win32_LogicalDisk -Filter "DriveType=3" }
```

-  **Note:** Your report must include each computer's name, each drive's letter, and each drive's free space and total size in bytes.

3. To produce an HTML report containing the results of the previous command, type the following command, and then press Enter:

```
Invoke-Command -Session $computers -ScriptBlock { Get-WmiObject -Class  
Win32_LogicalDisk -Filter "DriveType=3" } |  
ConvertTo-Html -Property PSComputerName,DeviceID,FreeSpace,Size
```

► **Task 4: Close all open PSSessions**

- To close all open PSSessions, type the following command, and then press Enter:

```
Get-PSSession |  
Remove-PSSession
```

Results: After completing this exercise, you should have performed several management tasks against multiple computers.

► **Task 5: Prepare for the next module**

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

- On the host computer, start **Hyper-V Manager**.
- In Microsoft Hyper-V Manager, in the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
- In the **Revert Virtual Machine** dialog box, click **Revert**.
- Repeat steps 2 and 3 for **10961C-LON-CL1**.

Studijní materiály Okškolení

Module 11: Using background jobs and scheduled jobs

Lab: Using background jobs and scheduled jobs

Exercise 1: Starting and managing jobs

► **Task 1: Start a Windows PowerShell remote job**

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. Click **Start**, and then type **powershell**.
3. In the results list, right-click **Windows PowerShell**, and click **Run as administrator**.
4. To start a Windows PowerShell remote job that retrieves a list of physical network adapters from **LON-DC1** and **LON-SVR1**, run:

```
Invoke-Command -ScriptBlock { Get-NetAdapter -Physical } -ComputerName LON-DC1,LON-SVR1 -AsJob -JobName RemoteNetAdapt
```

5. To start a Windows PowerShell remote job that retrieves a list of Server Message Block (SMB) shares from **LON-DC1** and **LON-SVR1**, type the following command, and then press Enter:

```
Invoke-Command -ScriptBlock { Get-SMBShare } -ComputerName LON-DC1,LON-SVR1 -AsJob -JobName RemoteShares
```

6. To start a Windows PowerShell remote job that retrieves all instances of the **Win32_Volume** class from every computer in Active Directory Domain Services, type the following command, and then press Enter:

```
Invoke-Command -ScriptBlock { Get-CimInstance -ClassName Win32_Volume } -ComputerName (Get-ADComputer -Filter * | Select -Expand Name) -AsJob -JobName RemoteDisks
```

Because some of the computers in the domain might not be online, this job might not complete successfully. That is expected.

► **Task 2: Start a local job**

1. To start a local job that retrieves all entries from the Security event log, type the following command, and then press Enter:

```
Start-Job -ScriptBlock { Get-EventLog -LogName Security } -Name LocalSecurity
```

2. To start a local job that produces 100 directory listings, type the following command, and then press Enter:

```
Start-Job -ScriptBlock { 1..100 | ForEach-Object { Dir C:\ -Recurse } } -Name LocalDir
```

This job is expected to take a very long time to complete. Do not wait for it to complete—proceed to the next task.

► Task 3: Review job status

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. To display a list of running jobs, type the following command, and then press Enter:

```
Get-Job
```

3. To display a list of running jobs whose names start with **remote**, type the following command, and then press Enter:

```
Get-Job -Name Remote*
```

► Task 4: Stop a job

- To stop the **LocalDir** job, type the following command, and then press Enter:

```
Stop-Job -Name LocalDir
```

► Task 5: Retrieve job results

1. Run **Get-Job** until no jobs have a status of **Running**.
2. To receive the results of the **RemoteNetAdapt** job, type the following command, and then press Enter:

```
Receive-Job -Name RemoteNetAdapt
```

3. To receive the results of the **RemoteDisks** job, type the following command, and then press Enter:

```
Get-Job -Name RemoteDisks -IncludeChildJob | Receive-Job
```

 **Note:** You will receive an error on the LON-CL1 data collection due to Windows PowerShell Remoting not being turned on. This is expected.

Results: After completing this exercise, you will have started jobs using two of the basic job types.

Exercise 2: Creating a scheduled job

► Task 1: Create job options

1. Ensure that you are signed into **LON-CL1** as **ADATUM\Administrator** with the password **Pa55w.rd**.
2. To create a new job option, type the following command, and then press Enter:

```
$option = New-ScheduledJobOption -WakeToRun -RunElevated
```

► Task 2: Create a job trigger

- To create a new job trigger, type the following command, and then press Enter:

```
$trigger1 = New-JobTrigger -Once -At (Get-Date).AddMinutes(5)
```

► **Task 3: Create a scheduled job and retrieve results**

- To register the job, type the following command, and then press Enter:

```
Register-ScheduledJob -ScheduledJobOption $option
-Trigger $trigger1
-ScriptBlock { Get-EventLog -LogName Security }
-MaxResultCount 5
-Name LocalSecurityLog
```

- To display a list of job triggers, type the following command, and then press Enter:

```
Get-ScheduledJob -Name LocalSecurityLog |
Select -Expand JobTriggers
```

Write down the time shown.

- Wait until the time returned in the output of step 2 has passed.
- To display a list of jobs, type the following command, and then press Enter:

```
Get-Job
```

- To receive the job results, run:

```
Receive-Job -Name LocalSecurityLog
```

You should see a list of log entries.

► **Task 4: Use a Windows PowerShell script as a scheduled task**

- On **LON-DC1**, in Server Manager, click **Tools**, and then select **Active Directory Users and Computers**.
- In the **Active Directory Users and Computers** console tree, select and expand **Adatum.com**, and then select **Managers**.
- In the details pane of **Managers**, select one of the user accounts. Right-click the account, and then select **Disable Account**. Click **OK** and then minimize **Active Directory Users and Computers**.
- Click **Start** and then type **Task Scheduler**, and then click **Task Scheduler** from the content menu.
- In the **Task Scheduler**, in the console tree, right-click **Task Scheduler (local)** and select **Create Task**.
- In the **Create Task** window, in the **General** tab, in the **Name** and **Description** text boxes, type **Delete Disabled User from Managers Security Group**. In the **Security options**, select the **Run whether user is logged on or not**, and then select the **Run with highest privileges** check box.
- In the **Triggers** tab, click the **New** button, and in the **New Trigger** window, under **Settings**, select **Daily**, and then in the **Start** time text box, change the time to *5 minutes from the current time*, and then click **OK**. If you get a **Task Scheduler** popup window click **OK**.
- In the **Actions** tab, click the **New** button, and in the **New Action** window, in the **Program/script** text box, type **PowerShell.exe**.
- In the **Add arguments (optional):** text box, type **-ExecutionPolicy Bypass E:\Labfiles\Mod11\DeleteDisabledUserManagersGroup.ps1**, click **OK** and in the pop-up window, select **Yes**.
- In the **Conditions** tab, review the items, but make no changes.
- In the **Settings** tab, at the bottom of the window, under **If the task is already running, then the following rule applies:** click the drop-down list and select **Stop the existing instance**. Then click **OK**.

12. In the **Task Scheduler** credentials pop-up, in the **Password** text box, type **Pa55w.rd**, and then click **OK**.
13. In the **Task Scheduler**, select **Task Schedule Library** and then in the upper details pane, select the **Delete Disabled User from Managers Security Group** item, and then in the lower details pane, select the **History** tab. After the five minutes are up, click **Refresh** in the **Actions** pane. You should see an item with the **Task Category** of **Task completed**.
14. Maximize **Active Directory Users and Computers**. Double-click the user you disabled. Select the **Member of** tab. The user should no longer be a member of the Managers security group.

Results: After completing this exercise, you will have created and run a scheduled job, and retrieved the job's results. You will also have a scheduled task that uses a Windows PowerShell script to remove disabled users from a security group.

► Task 5: Prepare for the next module

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right-click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.

Module 12: Using advanced Windows PowerShell techniques

Lab: Practicing advanced techniques

Exercise 1: Creating a profile script

- ▶ Task 1: Create a profile script
 - A script that performs these tasks is located at:
E:\Mod12\Labfiles\10961C_Mod12_LabA_Ex1_LAK.txt

Results: After completing this exercise, you should have created a profile script.

Exercise 2: Verifying the validity of an IP address

- ▶ Task 1: Verify the validity of an IP address
 - A script that performs these tasks is located at:
E:\Mod12\labfiles\10961C_Mod12_LabA_Ex2_LAK.txt

Results: After completing this exercise, you should have created a script that verifies the validity of an IP address.

Exercise 3: Reporting disk information

- ▶ Task 1: Report disk information
 - A script that performs these tasks is located at:
E:\Mod12\labfiles\10961C_Mod12_LabA_Ex3_LAK.txt

Results: After completing this exercise, you should have created a script that reports disk space on a server.

Exercise 4: Querying NTFS permissions

- ▶ Task 1: Query NTFS permissions
 - A script that performs these tasks is located at:
E:\Mod12\labfiles\10961C_Mod12_LabA_Ex4_LAK.txt

Results: After completing this exercise, you will have created a module that you can use to query NTFS permissions.

Exercise 5: Creating user accounts with passwords from a CSV file

► Task 1: Create user accounts with a password from a CSV file

- A script that performs these tasks is located at:
E:\Mod12\labfiles\10961C_Mod12_LabA_Ex5_LAK.txt.

Results: After completing this exercise, you will have created a script that will create new user accounts from a CSV file.

► Task 2: Prepare for the end of the course

When you have finished the lab, revert the virtual machines to their initial state. To do this, perform the following steps:

1. On the host computer, start **Hyper-V Manager**.
2. In the **Virtual Machines** list, right click **10961C-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for **10961C-LON-CL1** and **10961C-LON-SVR1**.