Microsoft

Microsoft Official Course

AZ-040T00

Automating Administration with PowerShell

AZ-040T00

Automating Administration with PowerShell

1 http://www.microsoft.com/trademarks

MICROSOFT LICENSE TERMS

MICROSOFT INSTRUCTOR-LED COURSEWARE

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to your use of the content accompanying this agreement which includes the media on which you received it, if any. These license terms also apply to Trainer Content and any updates and supplements for the Licensed Content unless other terms accompany those items. If so, those terms apply.

BY ACCESSING, DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT ACCESS, DOWNLOAD OR USE THE LICENSED CONTENT.

If you comply with these license terms, you have the rights below for each license you acquire.

1. **DEFINITIONS.**
    1. "Authorized Learning Center" means a Microsoft Imagine Academy (MSIA) Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
    2. "Authorized Training Session" means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.
    3. "Classroom Device" means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center's training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
    4. "End User" means an individual who is (i) duly enrolled in and attending an Authorized Training Session or Private Training Session, (ii) an employee of an MPN Member (defined below), or (iii) a Microsoft full-time employee, a Microsoft Imagine Academy (MSIA) Program Member, or a Microsoft Learn for Educators – Validated Educator.
    5. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
    6. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.
    7. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics, or Microsoft Business Group courseware.
    8. "Microsoft Imagine Academy (MSIA) Program Member" means an active member of the Microsoft Imagine Academy Program.
    9. "Microsoft Learn for Educators – Validated Educator" means an educator who has been validated through the Microsoft Learn for Educators program as an active educator at a college, university, community college, polytechnic or K-12 institution.
    10. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
    11. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies.
    12. "MPN Member" means an active Microsoft Partner Network program member in good standing.

13. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
14. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
15. "Trainer" means (i) an academically accredited educator engaged by a Microsoft Imagine Academy Program Member to teach an Authorized Training Session, (ii) an academically accredited educator validated as a Microsoft Learn for Educators – Validated Educator, and/or (iii) a MCT.
16. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.

2. **USE RIGHTS.** The Licensed Content is licensed, not sold. The Licensed Content is licensed on a **one copy per user basis**, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

- 2.1 Below are five separate sets of use rights. Only one set of rights apply to you.

  1. **If you are a Microsoft Imagine Academy (MSIA) Program Member:**

     1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
     2. For each license you acquire on behalf of an End User or Trainer, you may either:

        1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User who is enrolled in the Authorized Training Session, and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
        2. provide one (1) End User with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
        3. provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content.

     3. For each license you acquire, you must comply with the following:

        1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
        2. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
        3. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End

User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified Trainers who have in-depth knowledge of and experience with the Microsoft technology that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Authorized Training Sessions,
6. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
7. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

2. **If you are a Microsoft Learning Competency Member:**
   1. Each license acquire may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
   2. For each license you acquire on behalf of an End User or MCT, you may either:
      1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Authorized Training Session and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware provided, **or**
      2. provide one (1) End User attending the Authorized Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
      3. you will provide one (1) MCT with the unique redemption code and instructions on how they can access one (1) Trainer Content.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. you will ensure that each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
      3. you will ensure that each End User provided with a hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each MCT teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified MCTs who also hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Authorized Training Sessions using MOC,
6. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
7. you will only provide access to the Trainer Content to MCTs.

3. **If you are a MPN Member:**
   1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
   2. For each license you acquire on behalf of an End User or Trainer, you may either:
      1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Private Training Session, and only immediately prior to the commencement of the Private Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
      2. provide one (1) End User who is attending the Private Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
      3. you will provide one (1) Trainer who is teaching the Private Training Session with the unique redemption code and instructions on how they can access one (1) Trainer Content.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. you will ensure that each End User attending an Private Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Private Training Session,
      3. you will ensure that each End User provided with a hard copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
      4. you will ensure that each Trainer teaching an Private Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Private Training Session,

5. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Private Training Sessions,
6. you will only use qualified MCTs who hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Private Training Sessions using MOC,
7. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
8. you will only provide access to the Trainer Content to Trainers.

4. **If you are an End User:**
For each license you acquire, you may use the Microsoft Instructor-Led Courseware solely for your personal training use. If the Microsoft Instructor-Led Courseware is in digital format, you may access the Microsoft Instructor-Led Courseware online using the unique redemption code provided to you by the training provider and install and use one (1) copy of the Microsoft Instructor-Led Courseware on up to three (3) Personal Devices. You may also print one (1) copy of the Microsoft Instructor-Led Courseware. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

5. **If you are a Trainer.**
   1. For each license you acquire, you may install and use one (1) copy of the Trainer Content in the form provided to you on one (1) Personal Device solely to prepare and deliver an Authorized Training Session or Private Training Session, and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Trainer Content. You may not install or use a copy of the Trainer Content on a device you do not own or control. You may also print one (1) copy of the Trainer Content solely to prepare for and deliver an Authorized Training Session or Private Training Session.
   2. If you are an MCT, you may customize the written portions of the Trainer Content that are logically associated with instruction of a training session in accordance with the most recent version of the MCT agreement.
   3. If you elect to exercise the foregoing rights, you agree to comply with the following: (i) customizations may only be used for teaching Authorized Training Sessions and Private Training Sessions, and (ii) all customizations will comply with this agreement. For clarity, any use of "customize" refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

- 2.2 **Separation of Components.** The Licensed Content is licensed as a single unit and you may not separate their components and install them on different devices.
- 2.3 **Redistribution of Licensed Content.** Except as expressly provided in the use rights above, you may not distribute any Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.
- 2.4 **Third Party Notices.** The Licensed Content may include third party code that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code are included for your information only.
- 2.5 **Additional Terms.** Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to your use of that respective component and supplements the terms described in this agreement.

3. **LICENSED CONTENT BASED ON PRE-RELEASE TECHNOLOGY.** If the Licensed Content's subject matter is based on a pre-release version of Microsoft technology ("**Pre-release**"), then in addition to the other provisions in this agreement, these terms also apply:

    1. **Pre-Release Licensed Content.** This Licensed Content subject matter is on the Pre-release version of the Microsoft technology. The technology may not work the way a final version of the technology will and we may change the technology for the final version. We also may not release a final version. Licensed Content based on the final version of the technology may not contain the same information as the Licensed Content based on the Pre-release version. Microsoft is under no obligation to provide you with any further content, including any Licensed Content based on the final version of the technology.

    2. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft technology, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its technology, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.

    3. **Pre-release Term.** If you are an Microsoft Imagine Academy Program Member, Microsoft Learning Competency Member, MPN Member, Microsoft Learn for Educators – Validated Educator, or Trainer, you will cease using all copies of the Licensed Content on the Pre-release technology upon (i) the date which Microsoft informs you is the end date for using the Licensed Content on the Pre-release technology, or (ii) sixty (60) days after the commercial release of the technology that is the subject of the Licensed Content, whichever is earliest ("**Pre-release term**"). Upon expiration or termination of the Pre-release term, you will irretrievably delete and destroy all copies of the Licensed Content in your possession or under your control.

4. **SCOPE OF LICENSE.** The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:

    - access or allow any individual to access the Licensed Content if they have not acquired a valid license for the Licensed Content,
    - alter, remove or obscure any copyright or other protective notices (including watermarks), branding or identifications contained in the Licensed Content,
    - modify or create a derivative work of any Licensed Content,
    - publicly display, or make the Licensed Content available for others to access or use,
    - copy, print, install, sell, publish, transmit, lend, adapt, reuse, link to or post, make available or distribute the Licensed Content to any third party,
    - work around any technical limitations in the Licensed Content, or
    - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation.

5. **RESERVATION OF RIGHTS AND OWNERSHIP.** Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property

laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content.

6. **EXPORT RESTRICTIONS.** The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.
7. **SUPPORT SERVICES.** Because the Licensed Content is provided "as is", we are not obligated to provide support services for it.
8. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon termination of this agreement for any reason, you will immediately stop all use of and delete and destroy all copies of the Licensed Content in your possession or under your control.
9. **LINKS TO THIRD PARTY SITES.** You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.
10. **ENTIRE AGREEMENT.** This agreement, and any additional terms for the Trainer Content, updates and supplements are the entire agreement for the Licensed Content, updates and supplements.
11. **APPLICABLE LAW.**
    1. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
    2. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.
12. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.
13. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT AND ITS RESPECTIVE AFFILIATES GIVES NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT AND ITS RESPECTIVE AFFILIATES EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**
14. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT, ITS RESPECTIVE AFFILIATES AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO US$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.**

This limitation applies to

- anything related to the Licensed Content, services, content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential, or other damages.

Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contre-façon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 $ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers; et.
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Revised April 2019

Contents

Module 0 Course introduction

About this course

About this course

Welcome to the *Automating Administration with PowerShell* course, which focuses on using PowerShell to automate the administration and management of Windows operating systems and some of the most common services in Microsoft Azure and Microsoft 365. You'll learn to identify the commands you require to perform specific tasks and then build those commands. You'll also learn how to build scripts to accomplish advanced tasks such as automating repetitive tasks and generating reports.

Note: This course covers the use of PowerShell for managing Windows operating systems and Azure and Microsoft 365 services. It doesn’t cover management of the Linux or macOS operating systems.

Level: Intermediate

Audience

This course is for IT Professionals who are already experienced in general Windows Server, Windows client, Azure, and Microsoft 365 administration, and who want to learn more about using PowerShell for administration. No prior experience with any version of PowerShell or any scripting language is assumed. This course is also suitable for IT Professionals already experienced in server administration, including Microsoft Exchange Server, Microsoft SharePoint Server, and Microsoft SQL Server.

Prerequisites

This course assumes you have skills and experience with the following technologies and concepts:

- Windows Server administration, maintenance, and troubleshooting
- Windows Client administration, maintenance, and troubleshooting
- Basic security best practices
- Windows networking technologies and implementation

- Core networking technologies such as IP addressing, name resolution, and Dynamic Host Configuration Protocol (DHCP)

Labs and demonstrations

You'll perform the labs and demonstrations on a virtual lab environment from an authorized lab hoster.

Course syllabus

The course content includes a mix of content, demonstrations, hands-on labs, and reference links.

	Module Name
0	Course introduction
1	Getting started with Windows PowerShell
2	Windows PowerShell for local systems administration
3	Working with the Windows PowerShell pipeline
4	Using PSProviders and PSDrives
5	Querying management information by using CIM and WMI
6	Working with variables, arrays, and hash tables
7	Windows PowerShell scripting
8	Administering remote computers with Windows PowerShell
9	Managing Azure resources with PowerShell
10	Managing Microsoft 365 services with PowerShell
11	Using background jobs and scheduled jobs

Course resources

There are many resources that can help you learn about PowerShell. We recommend that you bookmark the following websites:

- **Microsoft Learn:**[1] Free role-based learning paths and hands-on experiences for practice.
- **PowerShell Documentation:**[2] Articles and how-to guides about using Windows Server.
- **Azure PowerShell documentation:**[3] Articles and sample scripts for using Azure PowerShell.
- **Get started with PowerShell for Microsoft 365:**[4] Information on using PowerShell to manage Microsoft 365

1 https://aka.ms/Microsoft-learn-home-page
2 https://aka.ms/azure-powershell-documentation
3 https://aka.ms/azure-powershell-documentation
4 https://aka.ms/get-started-with-powershell-for-microsoft-365

Module 1 Getting started with Windows PowerShell

Windows PowerShell overview

Lesson overview

You can use Windows PowerShell more easily and effectively by first understanding its background and intended use. In this lesson, you'll learn about PowerShell and its various versions. You'll also learn how to open and configure commonly used host applications such as the Windows PowerShell console and Windows PowerShell Integrated Scripting Environment (ISE). Finally, you'll learn to use Microsoft Visual Studio Code (VS Code) to develop PowerShell scripts.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe Windows PowerShell.
- Describe the major versions of Windows PowerShell.
- Identify the common Windows PowerShell hosting applications.
- Describe points to consider when using 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.
- Describe how to use VS Code for PowerShell scripting.

Windows PowerShell introduction

PowerShell is an automation solution that consists of a command-line shell, a scripting language, and a configuration-management framework.

Command-line shell

Windows PowerShell superseded the Windows command-line interface (**cmd.exe**) and the limited functionality of its batch file scripting language. PowerShell accepts and returns .NET objects and includes:

- A command-line history.
- Tab completion and prediction.
- Support for command and parameter aliases.
- Chaining commands that use the Pipeline feature.
- A robust in-console help system

Initially, Windows PowerShell was a platform built on the .NET Framework and only worked on Windows operating systems. However, with its recent releases, PowerShell uses the .NET Core and can run on Windows, MacOS, and Linux platforms. Due to their multi-platform support, these recent releases are referred to as PowerShell (rather than Windows PowerShell).

A scripting language

Commands provide 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. Using commands enables you to create custom solutions to complex administrative problems. Alternatively, you can simply run commands directly within the PowerShell console to complete a single task. The console is the CLI for PowerShell and is the primary way in which you'll interact with PowerShell.

Cmdlets use a Verb-Noun naming convention. For example, you can use the **Get-Command** cmdlet to list all cmdlets and functions that are registered in the command shell. The verb identifies the action for the cmdlet to perform, and the noun identifies the resource on which the cmdlet will perform its action.

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 PowerShell. In many cases, even when the application provides a graphical user interface (GUI) to manage a specific functionality, it relies on PowerShell to implement at least some of its features behind the scenes.

Configuration management framework

PowerShell incorporates the PowerShell Desired State Configuration (DSC) management framework. This enables you to manage enterprise infrastructure with code to help with:

- Using declarative configurations and repeatable scripts for repeatable deployments.
- Enforcing configurations settings and identifying when configuration drift takes place from standard requirements.
- Deploying configuration settings using push or pull models.

Applications and services with 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.

Windows PowerShell versions

As you learn about PowerShell, it's important to understand the various versions that you might encounter, depending upon your operating system (OS) type and edition. There are two main PowerShell platforms:

- Windows PowerShell
- PowerShell (originally referred to as PowerShell Core)

Windows PowerShell

Windows PowerShell is available exclusively for the Windows OS. Windows PowerShell 1.0 was introduced in 2006 as a component installable on Windows XP Service Pack 2 (SP2), Windows Server 2003 SP1, and Windows Vista. It was also an optional component of Windows Server 2008. In 2009, PowerShell 2.0 was integrated into Windows 7 and Windows Server 2008 R2. All versions of Windows PowerShell up to and including 5.1, which is the version available with Windows 10, are integrated with a Windows OS.

Windows PowerShell is an OS component, so it receives the same lifecycle support and licensing agreements as its parent OS.

PowerShell

PowerShell is shipped, installed, and configured separately from Windows PowerShell. First released as PowerShell Core 6.0 in 2018, it was the first version that offered multi-platform support, extending its availability to MacOS and Linux operating systems.

Note: The latest version of PowerShell is PowerShell 7.2, available via Microsoft Update.

PowerShell and Windows PowerShell are separately installed and you can run supported commands using either environment.

Standard Windows licensing agreements don't include PowerShell. Rather, it's supported under Microsoft paid support, Microsoft Enterprise Agreements, and Microsoft Software Assurance. Community support is also available.

Version release history

The following table provides a general timeline of the major PowerShell releases:

Table 1: PowerShell release timelines

Version	Release Date	Notes
PowerShell 7.2	November 2021	Built on .NET 6.0.
PowerShell 7.1	November 2020	Built on .NET 5.0.
PowerShell 7.0	March 2020	Built on .NET Core 3.1.
PowerShell 6.0	September 2018	Built on .NET Core 2.0. First release that's installable on Windows, Linux, and macOS.

Version	Release Date	Notes
PowerShell 5.1	August 2016	Released in Windows 10 Anniversary Update and Windows Server 2016 and as part of Windows Management Framework (WMF) 5.1.
PowerShell 5.0	February 2016	Integrated in Windows 10 version 1511. Released in Windows Management Framework (WMF) 5.0. Can be installed on Windows Server 2008 R2, Windows Server 2012, Windows 10, Windows 8.1 Enterprise, Windows 8.1 Pro, and Windows 7 SP1.
PowerShell 4.0	October 2013	Integrated in Windows 8.1 and Windows Server 2012 R2. Can be installed on Windows 7 SP1, Windows Server 2008 SP1, and Windows Server 2012.
PowerShell 3.0	October 2012	Integrated in Windows 8 and Windows Server 2012. Can be installed on Windows 7 SP1, Windows Server 2008 SP1, and Windows Server 2008 R2 SP1.
PowerShell 2.0	July 2009	Integrated in Windows 7 and Windows Server 2008 R2. Can be installed on Windows XP SP3, Windows Server 2003 SP2, and Windows Vista SP1.
PowerShell 1.0	November 2006	Installable on Windows XP SP2, Windows Server 2003 SP1, and Windows Vista. Optional component of Windows Server 2008.

Note: Throughout this module, topics will relate to both the latest Windows PowerShell and PowerShell versions (5.1 and 7.2). Most cmdlets will work using either platform. However, there'll be a note if a specific feature is only supported or relates to one specific platform.

Additional Reading: To support more recent versions of PowerShell on down-level operating systems, you might need to install the latest version of the Windows Management Framework. For more information, refer to [Install and Configure WMF 5.1] (https://aka.ms/install-and-configure-WMF-5.1).

Windows PowerShell applications

To interact with PowerShell, you use an application that embeds, or hosts, the PowerShell's engine. Some of those applications will offer a graphical user interface (GUI), such as Windows Admin Center or Exchange Admin Center in Microsoft Exchange Server. Both of them utilize PowerShell to carry out administrative tasks.

In this course, you'll primarily interact with Windows PowerShell through one of the two primary hosts that Microsoft provides:

- The Windows PowerShell console
- The Windows PowerShell Integrated Scripting Environment (ISE)

Windows PowerShell console

The console uses the Windows built-in console host application. This is similar to the command-line experience that the traditional Windows Command Prompt shell (**cmd.exe**) offers. However, all currently supported versions of Windows provide an updated console with more functionality, including syntax coloring. The console provides the broadest Windows PowerShell functionality, with a number of significant improvements introduced in Windows PowerShell 5.1.

Windows PowerShell ISE

The ISE is a Windows Presentation Foundation (WPF) application that provides rich editing capabilities, including IntelliSense code hinting and completion. In this course, you'll begin by using the console. This is an effective way to reinforce important foundational skills. Toward the end of the course, you'll shift to using the ISE as you begin to link multiple commands together into scripts.

Third parties offer other Windows PowerShell host applications. Several companies produce free and commercial Windows PowerShell scripting, editing, and console hosts. However, this module focuses on the host applications that the Windows OS provides.

Note: Windows PowerShell ISE only supports Windows PowerShell versions up to and including 5.1. It doesn't support subsequent versions of PowerShell (6.x or 7.x). You can use Microsoft Visual Studio Code with the PowerShell extension if you are looking for a similar scripting environment as the one that Windows PowerShell ISE provides.

Considerations when using PowerShell

As you use PowerShell to manage and automate management tasks, you take into account several considerations, including:

- Installing and using PowerShell side-by-side with Windows PowerShell.
- Running PowerShell using administrative credentials.
- Identifying and modifying the execution policy in PowerShell.

Installing and using PowerShell side-by-side with Windows PowerShell

Depending on Windows operating system versions and editions, organizations might have computers running different versions of PowerShell. Sometimes, these mixed-version environments are the result of organizational policies that don't permit the installation of newer PowerShell versions. They might also be the result of software products, especially server software, which have a dependency on a specific PowerShell version.

If you install the latest version of PowerShell, you end up with multiple PowerShell versions installed on your system. For example, PowerShell 7 is designed to coexist with Windows PowerShell 5.1 and will install to a new directory and enable side-by-side execution with Windows PowerShell.

Installing the latest version of PowerShell results in the following when compared to Windows PowerShell:

- *Separate installation path and executable name*. Windows PowerShell 5.1 is installed in the `$env:WINDIR\System32\WindowsPowerShell\v1.0` location. PowerShell 7 is installed in the `$env:ProgramFiles\PowerShell\7` location. The new location is added to your PATH, which allows you to run both Windows PowerShell 5.1 and PowerShell 7. In Windows PowerShell, the PowerShell executable is named `powershell.exe`. In version 6 and newer, the executable is named `pwsh.exe`. The new name makes it easy to support side-by-side execution of both versions.
- *Separate PSModulePath*. By default, Windows PowerShell and PowerShell 7 store modules in different locations. PowerShell 7 combines those locations in the `$Env:PSModulePath` environment variable. When importing a module by name, PowerShell checks the location that `$Env:PSModulePath` specifies. This allows PowerShell 7 to load both Core and Desktop modules.
- *Separate profiles for each version*. A PowerShell profile is a script that runs when PowerShell starts. This script customizes the PowerShell environment by adding commands, aliases, functions, variables, modules, and PowerShell drives. In Windows PowerShell 5.1, the profile's location is `$HOME\Documents\WindowsPowerShell`. In PowerShell 7, the profile's location is `$HOME\Documents\PowerShell`.
- *Separate event logs*. Windows PowerShell and PowerShell 7 log events to separate Windows event logs.

When you're reviewing a PowerShell session, it's important to determine which version you're using. To determine the current version, enter `$PSVersionTable` in the PowerShell console, and then select Enter. PowerShell displays the version numbers for various components, including the main PowerShell version number.

Running PowerShell using Administrative credentials

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

On 32-bit operating systems, PowerShell’s host applications are available only in 32-bit versions. When working with Windows PowerShell, the icons and window title bars don't have 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, if you intend to use PowerShell to perform administrative tasks, you might have to take an extra step to run PowerShell cmdlets with full administrative credentials. To do this, right-click or activate the context menu for the application icon, and then select **Run as Administrator**. When you're running PowerShell with administrative credentials, the host application’s window title bar will include the **Administrator** prefix.

Identifying and modifying the execution policy in PowerShell

The execution policy in PowerShell is meant to minimize the possibility of a user unintentionally running PowerShell scripts. You can think of it as a safety feature that controls the conditions under which Power-

Shell loads configuration files and runs scripts. This feature helps prevent the execution of malicious scripts.

Important: The execution policy in PowerShell isn't a security system that restricts user actions. For example, if users can't run a script, they can easily bypass a policy by entering the script contents at the command line.

To identify the effective execution policy for the current PowerShell session, use the following cmdlet:

```
Get-ExecutionPolicy
```

You can configure the following policy settings:

- **AllSigned**. Limits script execution on all signed scripts. This requires that all scripts are signed by a trusted publisher, including scripts that you write on the local computer. It prompts you before running scripts from publishers that you haven't yet classified as trusted or untrusted. Be aware that verifying the signature of a script of does not eliminate the possibility of that script being malicious. It simply provides an additional check that minimizes this possibility.
- **Default**. Sets the default execution policy, which is **Restricted** for Windows clients and **Remote-Signed** for Windows servers.
- **RemoteSigned**. Is the default execution policy for Windows server computers. Scripts can run, but the policy requires a digital signature from a trusted publisher on scripts and configuration files that have been downloaded from the internet. This setting doesn't require digital signatures on scripts that are written on the local computer.
- **Restricted**. Is the default execution policy for Windows client computers. It permits running individual commands, but it doesn't allow scripts.
- **Unrestricted**. Is the default execution policy for non-Windows computers, which you can't change. It allows unsigned scripts to run. This policy warns the user before running scripts and configuration files that aren't from the local intranet zone.
- **Undefined**. Indicates that there isn't an execution policy set in the current scope. If the execution policy in all scopes is **Undefined**, the effective execution policy is **Restricted** for Windows clients and **RemoteSigned** for Windows Server.

To change the execution policy in PowerShell, use the following command:

```
Set-ExecutionPolicy -ExecutionPolicy <PolicyName>
```

Configuring the PowerShell console

When you work in the console regularly, you might want to customize its settings by specifying custom preferences. These preferences include, for example, font size and type, the size and position of the console window and the screen buffer, the color scheme, and keyboard shortcuts.

Font type, color, and size

If the font size and color make it difficult for you to review content, 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 code. To determine whether this is the case, when you first open the console, enter 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 you find it difficult to differentiate between these characters, try changing the font. To change the font, select the **PowerShell** icon in the console window. From the shortcut menu, select **Properties**, and then select the **Font** tab. Select a new font style and size. TrueType fonts, which the double-T symbol indicate, are usually easier to review than Raster fonts. The PowerShell **Properties** dialog box provides a preview pane in which you can review the results of your choice.

Screen buffer and console window sizes

You can customize the console window's size and location. When setting the window size, you should understand the relationship between the **Screen Buffer Size** and **Windows Size** options, so that you don't 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'll 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 close to your actual screen's width so you maximize the amount of horizontal text you can display. This will prove useful later when you format the on-screen output that your scripts return.

The height of your screen buffer doesn't have to match your screen height. In most cases, it's much larger. If you set this value to a large number, you'll be able to scroll through large numbers of commands that are entered in a session or large amounts of output.

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

Color scheme

Finally, if you want to change the default 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 pop-up windows. You can't 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 there isn't a horizontal scroll bar. A vertical scroll bar will still display in the console.

Copying and pasting

The console host supports copying and pasting to and from the clipboard. It also supports using standard keyboard shortcuts, though they might not always work. This could be because of 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 select Enter to copy that text to the clipboard. Right-click or activate the context menu to paste. Starting with Windows 10 and Windows Server 2016, the Ctrl+C and Ctrl+V keyboard shortcuts also work for copy and paste.

Demonstration: Configuring the console

In this demonstration, you'll learn 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 doesn't display a horizontal scroll bar.
5. Start a shell transcript, and then name the file **Day1.txt**.
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 graphical environment that provides a script editor, a debugger, an interactive console, and several auxiliary tools that help you discover and learn Windows PowerShell commands. This module provides a basic overview on how the ISE works. You will learn about more advanced ISE's capabilities in the subsequent modules.

Panes

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

Customizing the view

The ISE provides several ways to customize the view. You can change the active font size by using the slider in the window's lower-right area. 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 also supports creation of visual themes. A *theme* is a collection of font and color settings that you can apply as a group to customize the ISE's appearance. There are several built-in themes intended for designated use cases, 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.

Note: The Windows PowerShell ISE is available on Windows OS, but it won't work with PowerShell 6 and newer. For a similar experience, you can use Visual Studio Code (VS Code) with the PowerShell extension. You will learn more about it in the next unit of this module.

Demonstration: Configuring the ISE

In this demonstration, you'll learn 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 Windows PowerShell console?

Using Visual Studio Code with PowerShell

Visual Studio Code (VS Code) is a Microsoft script editor that provides a rich and interactive script editing experience. This experience is very similar to the PowerShell Integrated Scripting Environment (ISE) when you use it with the PowerShell extension. VS Code supports the following PowerShell versions:

- PowerShell 7 and newer for Windows, macOS, and Linux

- PowerShell Core 6 for Windows, macOS, and Linux
- Windows PowerShell 5.1 for Windows operating systems only

Note: Visual Studio Code is different from Visual Studio. You can download VS Code from **the Visual Studio Code download page**[1] and the PowerShell extension from the Visual Studio Marketplace at **Visual Studio Marketplace**[2]. You can also install the extension directly from within VS Code.

ISE Mode in Visual Studio Code

After you install VS Code and the PowerShell extension, you can replicate the ISE experience by turning on **ISE Mode**.

Enabling ISE Mode makes several notable changes to the way VS Code works, including:

- Mapping keyboard functions in VS Code so they match those used in the ISE.
- Allowing you to replicate the VS Code user interface to resemble the ISE.
- Enabling ISE-like tab completion.
- Providing several ISE themes to make the VS Code editor look like the ISE.

Additional information: For more information on using Visual Studio Code and how to replicate the ISE experience, refer to **Using Visual Studio Code for PowerShell Development**[3].

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

What's the difference between Windows PowerShell and PowerShell Core?

Question 2

You want to start PowerShell 7 from the command line. Which command will you run?

Question 3

Why might you decide to use the ISE versus the console host?

Question 4

Describe how Visual Studio Code may help when you're working with PowerShell.

1 https://aka.ms/visual-studio-code-2
2 https://aka.ms/visual-studio-marketplace
3 https://aka.ms/using-visual-studio-code-for-powershell-development

Understand Windows PowerShell command syntax

Lesson overview

In this lesson, you'll learn about the cmdlet structure and parameters for using Windows PowerShell cmdlets. You'll also learn how to use tab completion and how to display **About** files content.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe cmdlet structure.
- Identify how to use Windows PowerShell parameters.
- Explain how to use tab completion.
- Explain how to display the **About** files 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 is virtually impossible. Fortunately, cmdlet creators build cmdlets by using a common format that helps you predict both a cmdlet's name and its syntax. This makes it much easier to discover and use cmdlets.

Note: The common format that PowerShell cmdlets use is the *Verb-Noun* notation.

Cmdlet verbs

The verb portion of a 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.

This list represents just some of the verbs that cmdlets use. Additionally, 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 retrieves information that a resource contains, such as a text file's content, whereas the **Get** verb retrieves the actual file.

Cmdlet nouns

The noun portion of a 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 for cmdlets that work with Windows services and the **Process** noun is 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 cmdlets begin with the prefix **Az**.

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. However, for now, you should consider them as all working in the same way. This module uses the terms *command* and *cmdlet* interchangeably.

Parameters

Parameters modify the actions that a cmdlet performs. You can specify no parameters, one parameter, or many parameters for a cmdlet.

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're passing contains spaces, you'll need to wrap the text in quotation marks. Some parameters accept multiple values, which you must separate by commas and no spaces.

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 it. For example, if you run the command **Get-Item**, you'll receive the following message from Windows PowerShell, which includes a prompt to provide a value for the *-Path* parameter:

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

If you enter the text **C:\** at the prompt and then press the Enter key twice, the command will run successfully. You must press the Enter key twice because this parameter can accept multiple values. Windows PowerShell will continue prompting for a new value until you press the Enter key without actually providing it.

In some cases, entering the parameter name is optional and you can just enter the parameter's value. If you run the command **Get-ChildItem C:\**, it's 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. You'll notice these throughout this course. Omitting the parameter name only works when a parameter position has been defined. Not all commands have positional parameters.

Switches

Switches are a special case. They're 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 all of its subdirectories. Without the **-Recurse** switch, only the items in the C:\ directory are returned.

Tab completion

Tab completion is a PowerShell feature that improves the speed and ease of finding and entering cmdlets and parameters. To use it, enter a few characters of a cmdlet or a parameter in the ISE or console, and then press the Tab key. PowerShell will automatically provide the missing part of the name for you based on the match on the characters you entered. If there are multiple matches, just press the Tab key multiple times until you are presented with the one you intended to use. This works for cmdlets and parameters, variable names, object properties, and file paths.

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 miss or reverse letters when entering the name such as **Get-DnsServerResponseRateLimitingExceptionList**.

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 the Tab key to review the available options. For parameters, just enter a dash (-) and you can press the Tab key multiple times to review all parameters for a cmdlet.

Tab completion even works with wildcards. If you know you want a cmdlet that operates on services, but aren't sure which one you want, enter the text ***-service** in the console, and then press the Tab key to review all cmdlets that contain the text **-service** in their names.

About files

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

You can review a complete list of these topics by running **Get-Help about***, and then reviewing a single topic by running **Get-Help about_topicname**. An example is **Get-Help about_common_parameters**. These commands don't use the *–Example* or *–Full* parameters of the **Help** command. However, they're 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.

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

Demonstration: Using About Files

In this demonstration, you'll learn how to review topics from **About** help files.

Demonstration steps

1. Review a list of the topics that **About** help files return.

2. Review the **about_aliases** help topic.
3. Review the **about_eventlogs** help topic in a window.
4. Review a help topic that will explain how to make the console beep.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Describe the common structure that PowerShell cmdlets use.

Question 2

Which key can you use to complete or suggest parameters for a PowerShell command?

Question 3

You need to obtain additional information about a specific PowerShell cmdlet. Which type of PowerShell files will provide this information?

Find commands and get help in Windows PowerShell

Lesson overview

In this lesson, you'll learn how to find Windows PowerShell cmdlets that you can use to perform 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'll use these strategies to identify the cmdlets that you need to complete tasks and learn how to use those cmdlets.

You'll also learn how to use **Get-Help** to retrieve detailed information about a cmdlet and its parameters.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the relationship between modules and cmdlets.
- Identify options for finding cmdlets.
- Describe aliases.
- Use aliases.
- Explain how to use **Show-Command**.
- Explain how to use **Get-Help**.
- Review help.
- Explain how to interpret the help file contents.
- Explain how to update the local help content.

What are modules?

Modules are groups of related PowerShell capabilities that are bundled together into a single unit. For the purposes of this class, you can think of them as containers hosting multiple cmdlets. Modules help with organizing cmdlets into distributable units. Microsoft and other software companies provide modules as part of the management tools for their applications and services.

To use a module's cmdlets, the module must be loaded into the current PowerShell session. Typically this takes place automatically but, depending on your configuration, might require that you load modules explicity by running the **Import-Module** cmdlet. Some server products, such as Microsoft Exchange Server, provide a shortcut to what appears to be a dedicated management shell. However, this is really a normal PowerShell console session with application-specific modules already loaded.

Autoloading

In Windows PowerShell version 3.0 and newer, modules load automatically if you run a cmdlet that is part of that module. This works if the module that contains the cmdlet is in a folder under the module load paths. By default, these folders include **%systemdir%\WindowsPowerShell\v1.0\Modules** and **%userprofiles%\Documents\WindowsPowerShell\Modules**. The list of folders is stored in the `$env:PSModulePath` environment variable. When you explicitly import a module by name, PowerShell checks the locations referenced by that environment variable.

For PowerShell 7, the **PSModulePath** includes the following locations:

```
C:\Users\<user>\Documents\PowerShell\ModulesC:\Program Files\PowerShell\
ModulesC:\Program Files\PowerShell\7\ModulesC:\Program Files\WindowsPower-
Shell\ModulesC:\WINDOWS\System32\WindowsPowerShell\v1.0\Modules
```

Note: When using Windows PowerShell, the path **%systemdir%\WindowsPowerShell\v1.0\Modules** is commonly referred to by using the combination of the `$PSHome` environment variable (which points to **%systemdir%\WindowsPowerShell\v1.0**) and the **Modules** path (i.e. by using the `$PSHome\Modules` notation). For PowerShell 7.0, the `$PSHome` environment variable refers to **C:\Program Files\PowerShell\7**.

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. As you begin working with PowerShell, finding the cmdlet that you need might be a 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.

Like the **Get-Help** command, **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 returned results. 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 won't 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 get a list of commands that have nouns starting with **event** and that use the **Get** verb.

When you attempt to identify 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're trying to find a command that works with Windows event logs.

Using modules to discover cmdlets

When you use the **Get-Module** command, it displays a partial list of cmdlets that the module you reference contains. However, you can use the module in another way to find its cmdlets.

For example, if you've discovered the module **NetAdapter**, you would expect that it should contain cmdlets you can use to manage network adapters. You can find all applicable commands 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

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** performs a full-text search by using your query string if it can't 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 returned.

You can also refer to the **Related Links** section of a cmdlet that you know is related to the one you're searching for. This section of the help topic includes related cmdlets.

Finding cmdlets on the internet

You're not limited to searching for cmdlets that your computer already has installed. You can search the internet to find a wide variety of Microsoft and non-Microsoft modules and cmdlets. If you simply search by using the terms **PowerShell** and the technology you're working with, you'll find many links to articles on Microsoft 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.

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 similarly to 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.

Additional Reading: For more information about PowerShell Gallery, refer to **PowerShell Gallery**[4].

What are aliases?

If you have experience using the traditional Windows Command Prompt shell (**cmd.exe**), you're likely also 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. That's because behind the scenes, these commands are running native PowerShell cmdlets. The **dir** command runs **Get-ChildItem**, the **cd** command runs **Set-Location**, and the **mkdir** command runs **New-Item**. These commands work with PowerShell because they're *aliases* of the cmdlets that perform the equivalent action.

Aliases and parameters

It's important to note that aliases typically don't support the parameters that the original commands use. For example, if you run the command **dir /o:d** in the console, you'll receive an error because **Get-ChildItem** doesn't recognize the */o:d* parameter. Instead, you can use the *dir | sort LastAccessTime* to list the contents of the current folder sorted by last accessed date and time in the ascending order.

4 https://aka.ms/iast9g

Get-Alias

PowerShell includes more than just aliases for legacy batch and Linux commands. It also provides other aliases, such as **gci** for **Get-ChildItem**, which you can use to replace a full command with its abbreviated notation and minimize the amount of typing required. You can discover aliases, their definitions, and the commands that they run, by using the **Get-Alias** cmdlet. **Get-Alias** with no parameters returns 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*** returns aliases for both **diff** and **dir**.

You can also use the **Get-Alias** cmdlet to discover new cmdlets. For example, you use the batch command **del** to delete a file or folder. You can enter 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 aren't saved between Windows PowerShell sessions. You can use a Windows PowerShell profile to recreate the alias every time you open Windows PowerShell.

Additional Reading: For more information about creating and using a PowerShell profile, refer to **about_Profiles**[5].

Disadvantages of aliases

Aliases can help you enter commands more quickly, but they tend to make scripts harder to review 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 review. In most cases, using tab completion will make command entry almost as fast as entering an alias name and, at the same time, ensure its accuracy.

Demonstration: Using aliases

In this demonstration, you'll learn 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. Review the definition for the **dir** alias.

5 https://aka.ms/about-profiles

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. Display the definition for the **list** alias.
6. Display the various aliases for **Get-ChildItem**.

Using Show-Command

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

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

```
Show-Command –Name Get-ADUser
```

The –*Name* parameter is positional, so the following produces the same result:

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

If you select the **ShowCommand 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 are the same.

Note: In these examples, **Show-Command** is the command that you're actually running, but **Get-ADUs-er** is the name of the command that you want to review in the dialog box.

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

Once you provide values for all 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 review 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 predefined set of core capabilities. You'll learn more about many of the common parameters in upcoming modules. However, if you want to find out about them now, run **help about_common_parameters** in Windows PowerShell and review the results.

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 through it. You can also use the **Help** function or the **Man** alias, which maps to the **Get-Help** command. All three return basically the same results, including a short and long description of the cmdlet, the syntax, any additional remarks from the help author, and links to related cmdlets or additional information online. The **help** and **Man** commands display content in the console one page at a time. The ISE displays the entire help content.

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

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

Get-Help parameters

The **Get-Help** command accepts parameters that allow you find additional information beyond the information displayed by default. A common reason to seek additinal help is to identify usage examples for a command. Windows PowerShell commands commonly include many such examples. For instance, running the command **Get-Help Stop-Process –Examples** will provide 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 isn't 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 enter the parameter name (in which case **named** displays).
- Whether a parameter accepts pipeline input and, if so, how.

Other **Get-Help** parameters include:

- *-ShowWindow*. Displays the help topic in a separate window, which makes it much easier to access 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 don't know their names. For example, if you want all cmdlets that operate on processes, you can enter the command **Get-Help *process*** in the console, and then press the Enter key. The results match those returned by the command **Get-Command *process***, except that **Get-Help** displays a *synopsis*. This is a short description that helps you identify the command you want.

Sometimes, you might specify a wildcard search that doesn't match any command name. For example, running **Get-Help *beep*** won't find any commands that have *beep* in their name. When no results are found, the help system performs a full-text search of available command descriptions and synopses. This locates any help files that contain *beep*. If there is only a single file with a match, the help system displays its content, rather than displaying 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: Reviewing Help

In this demonstration, you'll learn how to use the help system's various options.

Demonstration steps

1. Display basic help for a command.
2. Display help in a floating window.
3. Display usage examples.
4. Display online help (if connected to the internet).

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 review the examples and try to understand their usage. However, it's uncommon for examples to cover all possible variations of that's command's usage. Learning to interpret the help-file syntax can help you to identify all capabilities of a given command.

Get-EventLog help

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

```
Get-EventLog [-LogName] <String> [[-InstanceId] <Int64[]>] [-After <Date-
Time>] [-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] [<CommonParame-
ters>]
```

The two blocks of text are *parameter sets*, each of which represents a 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 can't mix and match parameters between sets. That is, if you decide to use the *–List* parameter, you can't also use *–LogName*, because those two don't 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 depicts 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's a *positional parameter*. You can't run the command without a log name. However, you don't have to enter the *–LogName* parameter name. You do need to pass the log name string as the first parameter, because that's 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 can't omit every parameter. For example, the -*ComputerName* parameter can't have the parameter name omitted. Additionally, you can quickly lose track of what goes where. When you provide parameter names, the parameters can appear in any order, as the following command depicts:

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

However, when you omit parameter names, you must ensure to enter parameters in the correct order. For example, the following command won't work because the log name value does not appear in the correct 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 way to specify multiple values is by using a comma-separated list. You don't have to enclose the values in quotation marks, unless the values themselves contain a comma or 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 reviewing the command's full help. For example, run **Get-Help Get-EventLog –Full** to review the full help for **Get-EventLog**, and notice the additional information that displays. For example, you can confirm that the –*LogName* parameter is mandatory and appears in the first position.

Best Practice: If you're 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 review and troubleshoot, and they make it easier to notice mistakes if you're entering the command incorrectly.

Updating help

Windows PowerShell 3.0 and newer versions don't ship with help files. Instead, help files are available as an online service. Microsoft-authored commands have their help files hosted on Microsoft-owned web servers. Non-Microsoft commands are sometimes availalble online, as long as the author or vendor builds the module correctly and provides an online location for the help files. By using the online model, authors who write commands, including Microsoft authors, can make corrections and improvements to their help files over time, and then deliver them without having to create an entire product update.

Run **Update-Help** to scan your computer for all installed modules, retrieve online help locations for each, and try to download their respectvive help files. 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. Note that error messages will be displayed if help isn't 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 isn't 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 once every 24 hours, even if you run the command multiple times in a row. To override this behavior, include the *–Force* parameter.

The companion to **Update-Help** is **Save-Help**. It downloads the help content and saves it to a location that you specify. This allows you to copy that content to computers that aren't connected to the internet. **Update-Help** offers a parameter to specify an alternative source location. This makes it possible to update help on computers which are not connected to the internet.

Prior to Windows PowerShell 4.0, **Update-Help** and **Save-Help** download help only for the cmdlets installed on the local computer (where you run the command from). In Windows PowerShell 4.0 and newer, you can use **Save-Help** for modules installed on remote computers.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

You attempt to run a PowerShell cmdlet to create a new user in Active Directory, but receive an error that no PowerShell cmdlet is available to perform the task. What might you need to do first?

Question 2

How would you search for a cmdlet that retrieves a computer's properties from Active Directory?

Question 3

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

Module 01 lab and review

Lab: Configuring Windows PowerShell, and finding and running commands

Scenario

You're an administrator who's using Windows PowerShell to automate several 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.

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

Objectives

After completing this lab, you'll be able to:

- Open and configure the Windows PowerShell console application.
- Open and configure the Windows PowerShell ISE application.
- Find and run Windows PowerShell commands.
- Use Windows PowerShell Help and About topics to learn new shell concepts and techniques.

Estimated Time: 60 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

Which of the following PowerShell versions are integrated into Windows? Choose two.

☐ PowerShell 4.0

☐ PowerShell 5.1

☐ PowerShell 6.0

☐ PowerShell 7.0

☐ PowerShell 7.1

Question 2

True or False? For the PowerShell cmdlet `Get-EventLog`*, the verb portion of the cmdlet is the word* `EventLog`.

- ☐ True
- ☐ False

Question 3

You wish to join multiple computers to the Adatum domain. The `Add-Computer` *cmdlet's* `-ComputerName` *parameter accepts multiple values. Which of the following is a set of valid values for this parameter?*

- ☐ `-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`

Question 4

The `Get-ChildItem` *cmdlet is an alias for which command?*

- ☐ cd
- ☐ mkdir
- ☐ dir
- ☐ del
- ☐ rd

Question 5

You need to ensure that the latest PowerShell help files are installed on your local device. Which of the following is the correct PowerShell cmdlet to use?

- ☐ `Update-Help`
- ☐ `Invoke-Help`
- ☐ `Get-Help`
- ☐ `Register-Help`

Answers

Question 1

What's the difference between Windows PowerShell and PowerShell Core?

Windows PowerShell is a component of the Windows operating system. PowerShell Core is shipped, installed, and configured separately from Windows PowerShell. PowerShell Core is also a cross-platform component that supports MacOS and Linux operating systems.

Question 2

You want to start PowerShell 7 from the command line. Which command will you run?

You will run the pwsh.exe command to start PowerShell 7.

Question 3

Why might you decide to use the ISE versus the console host?

The ISE supports richer editing capabilities and can display a wider range of fonts. It's also compatible with double-byte character sets, making it compatible with a wider variety of written languages. However, if you're simply running a few commands, the console is sufficient.

Question 4

Describe how Visual Studio Code may help when you're working with PowerShell.

Visual Studio Code replicates the ISE experience, specifically for PowerShell Core. However, you must install the PowerShell extension to provide similar functionality as the ISE.

Question 1

Describe the common structure that PowerShell cmdlets use.

The common structure that PowerShell cmdlets use is the Verb-Noun format. The verb portion of the cmdlet name indicates what the cmdlet does. The noun portion of the cmdlet name indicates what kinds of resources or objects the cmdlet affects.

Question 2

Which key can you use to complete or suggest parameters for a PowerShell command?

You can use the Tab key to invoke Tab completion when entering a PowerShell command.

Question 3

You need to obtain additional information about a specific PowerShell cmdlet. Which type of PowerShell files will provide this information?

About files will provide additional information and examples for specific PowerShell cmdlets.

Question 1

You attempt to run a PowerShell cmdlet to create a new user in Active Directory, but receive an error that no PowerShell cmdlet is available to perform the task. What might you need to do first?

You first must ensure that the Active Directory module is loaded into PowerShell.

Question 2

How would you search for a cmdlet that retrieves a computer's properties from Active Directory?

You can use what you know about cmdlet name structures to help you guess a cmdlet's name. You know that Get retrieves resources so that you can work with their properties. You also know that the nouns associated with Active Directory have the prefix AD. Based on this information, you can use the Get-Help command and a wildcard value for the cmdlet name to search for possible cmdlets by running the command `Get-Help Get-AD*`. Because you're searching for the cmdlets that operate specifically on computers, you could even check if Get-Help Get-ADComputer returns results.

Question 3

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

Get-Help searches for help topics. Get-Command searches for commands. There should be a help topic for every cmdlet, but no one enforces this. As a result, Get-Help might not return anything for an existing command that doesn't have a help topic. Additionally, when no results return when you query a command name, Get-Help will perform a full-text search of the help files by using the same query value. Get-Command has no such capability. As a result, Get-Help might return results when Get-Command does not.

Question 1

Which of the following PowerShell versions are integrated into Windows? Choose two.

- [✓] PowerShell 4.0
- [✓] PowerShell 5.1
- ☐ PowerShell 6.0
- ☐ PowerShell 7.0
- ☐ PowerShell 7.1

Explanation
Only PowerShell 4.0 and 5.0 are included and integrated in Windows. You need to install PowerShell 6.0 and newer versions.

Question 2

True or False? For the PowerShell cmdlet `Get-EventLog`, the verb portion of the cmdlet is the word `EventLog`.

☐ True

■ False

Explanation
False is the correct answer. For the `Get-EventLog` *cmdlet, the word* `Get` *is the verb portion of the cmdlet.* `EventLog` *is the noun portion of the cmdlet.*

Question 3

You wish to join multiple computers to the Adatum domain. The `Add-Computer` cmdlet's `-ComputerName` parameter accepts multiple values. Which of the following is a set of valid values for this parameter?

☐ `-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`

Explanation
`-ComputerName LON-CL2,LON-CL3,LON-CL4` *is the correct answer. The correct way to pass multiple values to a parameter that accepts them is by separating them with a comma and no spaces. In option 1, a semicolon is not a valid separator for parameter values. In option 2, the entire string, including commas and spaces, will be passed as a single value. In option 3, spaces are not a valid separator for parameter values. In option 4, you don't pass multiple parameter values by specifying the parameter multiple times.*

Question 4

The `Get-ChildItem` cmdlet is an alias for which command?

☐ cd

☐ mkdir

■ dir

☐ del

☐ rd

Explanation
The `dir` *command is the equivalent for the PowerShell cmdlet* `Get-ChildItem`*.*

Question 5

You need to ensure that the latest PowerShell help files are installed on your local device. Which of the following is the correct PowerShell cmdlet to use?

■ `Update-Help`

☐ `Invoke-Help`

☐ `Get-Help`

☐ `Register-Help`

Explanation
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.*

Module 2 Windows PowerShell for local systems administration

Active Directory Domain Services administration cmdlets

Lesson overview

Active Directory Domain Services (AD DS) and its related services form the core of Windows Server–based networks. The AD DS database stores information about the objects that are part of the network environment, 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.

You can use the Active Directory module for Windows PowerShell to automate AD DS administration. By using Windows PowerShell for AD DS administration tasks, you can speed them up by making bulk updates instead of updating AD DS objects individually. If you don't have the Active Directory module installed on your machine, you'll need to download the correct Remote Server Administration Tools (RSAT) package for your operating system. Starting with Windows 10 October 2018 Update, RSAT is included as a set of Features on Demand. To install an RSAT package, you can refer to **Optional features** in **Settings** and select **Add a feature** to review the list of available RSAT tools. For AD DS administration, you'll need to select the **RSAT: Active Directory Domain Services and Lightweight Directory Services Tools** option.

Note: The Active Directory module for Windows PowerShell works for both Windows PowerShell and PowerShell.

In this lesson, you'll learn about the cmdlets used for administering AD DS. To find Active Directory cmdlets, search for the prefix "AD," which most Active Directory cmdlets have in the noun part of the cmdlet name.

Lesson objectives

After completing this lesson, you'll 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. You can use the Active Directory module for Windows PowerShell cmdlets to create, modify, and delete user accounts individually or in bulk. User account cmdlets have the word "User" or "Account" in the noun part of the name. To identify the available cmdlets, include them in wildcard name searches when you're using **Get-Help** or **Get-Command**.

The following table lists common cmdlets for managing user accounts.

Table 1: Cmdlets for user account management

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
Unlock-ADAccount	Unlocks a user account that's been locked after exceeding the permitted 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 review other properties, you'll 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 **anabowman** by entering the following command in the console, and then pressing the Enter key:

```
Get-ADUser -Identity anabowman -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 describe in more detail. For example, to retrieve all AD DS users and their properties, enter the following command in the console, and then press the Enter key:

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

Note: The help for **Get-ADUser** includes examples that use features of Windows PowerShell that you'll learn about later in this course. For example, more advanced *-Filter* operations require comparison operators, which you'll 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 don't use the *-AccountPassword* parameter, then no password is set, and the user account is disabled. You can't 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.

Table 2: Common parameters for New-ADUser

Parameter	Description
-AccountExpirationDate	Defines the expiration date for a user account
-AccountPassword	Defines the password for a user account
-ChangePasswordAtLogon	Requires a user account to change passwords at the next sign-in
-Department	Defines the department for a user account
-DisplayName	Defines the display name for a user account
-HomeDirectory	Defines the location of the home directory for a user account
-HomeDrive	Defines the drive letters that map to the home directory for a user account
-GivenName	Defines the first name of a user account

To add a user account and set its Department attribute to **IT** , enter the following command in the console, and then press the Enter key:

```
New-ADUser "Ana Bowman" -Department IT
```

Because no password was set by the command you ran, the user account is disabled, and you can't enable it until a password is assigned.

Note: Later in the course, you'll learn how to use comma-delimited files for bulk creation of user accounts with passwords automatically assigned.

Group management cmdlets

The management of Active Directory groups closely relates to the management of users. You can use the Active Directory module for Windows PowerShell cmdlets to create and delete groups and to modify group properties. You can also use these cmdlets to change the group membership.

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 common cmdlets for managing groups.

Table 1: Cmdlets for group management

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. 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**.

Table 2: Common parameters for New-ADGroup

Parameter	Description
-Name	Defines the name of a group
-GroupScope	Defines the scope of a group as **DomainLocal**, **Global**, or **Universal**; you must provide this parameter
-DisplayName	Defines the Lightweight Directory Access Protocol (LDAP) display name for an object
-GroupCategory	Defines whether a group is a security group or a distribution group; if you don't specify either, a security group is created
-ManagedBy	Defines a user or group that can manage a group

Parameter	Description
-Path	Defines the OU or container in which a group is created
-SamAccountName	Defines a name that is backward-compatible with older operating systems

For example, to create a new group named **FileServerAdmins**, enter the following command in the console, and then press the Enter key:

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

Managing group membership

As previously mentioned, 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.

***-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 add a user account as a member to a group.
- You cannot provide a list of groups to these cmdlets.
- You can *pipe* a list of members to these cmdlets.

Demonstration: Managing users and groups

In this demonstration, you'll learn 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
```

Cmdlets for managing computer objects

The Active Directory module for Windows PowerShell 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.

Table 1: Cmdlets for computer account management

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**.

Table 2: 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 example is a command that you can use to create a computer account:

```
New-ADComputer -Name LON-CL10 -Path "ou=marketing,dc=adatum,dc=com" -Ena-
bled $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 physical device or its operating system. For example, you can use the **Add-Computer** cmdlet to join a computer to a domain. To manage the properties of the physical computer and its operating system, use the *-**Computer** cmdlets.

OU management cmdlets

Windows PowerShell provides cmdlets that you can use to create, modify, and delete Active Directory Domain Services (AD DS) Organizational Units (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 cmdlets that you can use to manage OUs.

Table 1: Cmdlets for OU management

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 lists common parameters for the **New-ADOrganizationalUnit** cmdlet.

Table 2: Parameters for New-ADOrganizationalUnit

Parameter	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 example is 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'll sometimes need to manage Active Directory objects that don't 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.

***-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 cmdlets that you can use to manage Active Directory objects.

Table 1: 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

Cmdlet	Description
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**.

Table 2: 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 isn't 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 "AnaBowmancontact" -Type contact
```

Demonstration: Managing Active Directory objects

In this demonstration, you'll learn 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.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

You need to use PowerShell to review the Department and email address for an Active Directory user named User1. How will you format the PowerShell command?

Question 2

You need to create a Global security group named Group1. Which type of parameter will you use to specify a Global group when using the New-ADGroup PowerShell cmdlet?

Question 3

What is the default value for the -ProtectedFromAccidentalDeletion parameter of New-ADOrganizational-Unit cmdlet?

Network configuration cmdlets

Lesson overview

Both Windows PowerShell and PowerShell Core support cmdlets that you can use to manage all aspects related to network settings on Windows devices. Settings that you can configure with PowerShell include TCP/IP, Domain Name System (DNS), firewall, and routing table configurations. 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 traditional command-line interface tools such as **ping.exe** and **tracert.exe**, which are used to identify and diagnose network connectivity and configuration settings.

In this lesson, you'll learn about the PowerShell modules and cmdlets used for configuring network settings for Windows devices.

Lesson objectives

After completing this lesson, you'll be able to:

- Identify cmdlets for managing TCP/IP settings.
- Describe how to manage local routing table settings.
- Describe how to modify DNS client configuration.
- List cmdlets for managing Windows Firewall.
- Configure network settings.

Managing IP addresses

PowerShell includes the **NETTCPIP** module, which consists of TCP/IP-specific cmdlets used to manage network settings for Windows servers and devices. You can use the **NETTCPIP** cmdlets to add, remove, change, and validate IP address settings.

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 address settings.

Table 1: Cmdlets for managing IP address settings

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 address settings

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 the **New-NetIPAddress** cmdlet.

Table 2: Parameters for New-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" -Prefix-
Length 24 -DefaultGateway 192.168.1.1
```

The **New-NetIPAddress** cmdlet also accepts the *–AddressFamily* parameter, which defines either the IPv4 or IPv6 IP address family. If you don't use this parameter, the address family property is detected automatically.

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 **NETTCPIP** PowerShell module also includes cmdlets used to manage the routing table for Windows servers and devices.

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 and settings.

Table 1: Cmdlets for managing routing table entries and settings

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 on a Windows computer. The **New-NetRoute** cmdlet requires you to identify the network interface and destination prefix.

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

Table 2: Parameters for New-NetRoute

Parameter	Description
-DestinationPrefix	Defines the destination prefix of an IP route

Parameter	Description
-InterfaceAlias	Defines the network interface, by alias, for an IP route
-InterfaceIndex	Defines the network interface, by index, for an IP route
-NextHop	Defines the next hop for an IP route
-RouteMetric	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" -De-
faultGateway 192.168.1.1
```

Managing DNS clients

PowerShell offers cmdlets for managing DNS client settings, DNS name query resolution, and for securing DNS clients.

DNS client management cmdlets are part of the **DNSClient** PowerShell module and have the text “DnsClient” in the noun part of the name.

The following table lists common cmdlets for modifying DNS client settings.

Table 1: Cmdlets for modifying DNS client settings

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 inter-face

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

PowerShell supports the **NetSecurity** module that contains cmdlets to manage local Network Security configurations such as Windows firewall rules and IP security settings.

To manage firewall settings, use cmdlets that have the text “NetFirewall” in their names. For firewall rule management, use cmdlets that contain the noun “NetFirewallRule.”

The following table lists common cmdlets for managing firewall settings and rules.

Table 1: Cmdlets for managing firewall settings and rules

Cmdlet	Description
New-NetFirewallRule	Creates a new firewall rule

Cmdlet	Description
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 cmdlets:

- 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'll learn how to:

- Test the network connection to **LON-DC1**.
- Review 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

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

   Test-Connection LON-DC1

Review the network configuration for LON-CL1

- Run the following command:

  ```
  Get-NetIPConfiguration
  ```

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, enter the following command, and then press the Enter key:

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

3. Enter **Y** and then press the Enter key 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

1. Run the following command:

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

2. Enter **Y** and press the Enter key twice to confirm the change.
3. 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
  ```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

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

Question 2

You need to manage firewall settings and rules with PowerShell. Which PowerShell module contains the required cmdlets?

Server administration cmdlets

Lesson overview

PowerShell is commonly used to perform administration tasks for Windows Server features and services.

In this lesson, you'll learn about cmdlets used to configure settings related to Group Policy, Server Manager, Hyper-V, and Internet Information Services (IIS).

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the cmdlets for managing Group Policy Objects (GPOs).
- 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 and administering Internet Information Services (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 Active Directory Domain Services (AD DS) OUs, including setting GPO inheritance and permissions. Group Policy management cmdlets require Remote Server Administration Tools (RSAT) 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.

Note: The **GroupPolicy** module for Windows PowerShell hasn't been officially tested with PowerShell versions 6 or 7.x. It's recommended to use Windows PowerShell to run **GroupPolicy** module cmdlets.

Additional reading: For more information on PowerShell Core module compatibility, refer to **PowerShell 7 module compatibility**[1].

The following table lists common cmdlets for managing GPOs.

Table 1: Cmdlets for managing GPOs

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
Restore-GPO	Restores a GPO from backup files
New-GPLink	Links a GPO to an AD DS container

1 https://aka.ms/powershell-7-module-compatibility

Cmdlet	Description
Import-GPO	Imports GPO settings from a backed-up GPO
Set-GPRegistryValue	Configures one or more registry-based policy settings in a GPO

Creating a new 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 doesn't link the created GPO to an AD DS container. To link a GPO to a container, use the **New-GPLink** cmdlet.

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 organizational unit:

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

Server Manager cmdlets

The **ServerManager** module for 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."

Note: The **ServerManager** module cmdlets can only be targeted against and run on Windows Server operating systems. If you try to use these cmdlets on a Windows client-based operating system, you'll receive an error message.

The following table lists the server management cmdlets.

Table 1: Server management cmdlets

Cmdlet	Description
Get-WindowsFeature	Obtains and displays 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 following command installs network load balancing on the local server:

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

Hyper-V cmdlets

PowerShell offers more than 200 cmdlets for managing Hyper-V Virtual machines (VMs), virtual hard disks, and other components of a Hyper-V environment. Hyper-V cmdlets are available in the **Hyper-V** module for PowerShell.

The Hyper-V cmdlets are available when you install the **Hyper-V Management tools** feature on a Windows client operating system, or the **Hyper-V Module for Windows PowerShell** feature on Windows Server.

Hyper-V cmdlets use one of three prefixes:

- "VM" for virtual machine cmdlets
- "VHD" for virtual hard disk cmdlets
- "VFD" for virtual floppy disk cmdlets

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

Table 1: 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
Export-VM	Exports a VM to a file
Checkpoint-VM	Creates a checkpoint of a VM

IIS administration cmdlets

The Web server role includes Internet Information Services (IIS), which is commonly used to manage websites and web-based applications. IIS supports PowerShell cmdlets to allow you to configure and manage application pools, websites, web applications, and virtual directories.

IIS management cmdlets are available in the **IISAdministration** module for PowerShell and have the prefix "IIS" in the noun part of their names. Sites use the noun "IISSite".

To manage web-based applications, you can use the **WebAdministration** module for PowerShell, which includes cmdlets for managing web applications. These cmdlets use the noun "WebApplication". Cmdlets for managing application pools use the noun "WebAppPool."

Note: The WebAdministration module has mostly been replaced by updated features included in the **IISAdministration** module. For any IIS-related management tasks, it's recommended to use the **IISAdministration** module.

The following table lists common IIS and web application administration cmdlets.

Table 1: IIS and web application administration cmdlets

Cmdlet	Description
New-IISSite	Creates a new IIS website
Get-IISSite	Gets properties and configuration information about an IIS website
Start-IISSite	Starts an existing IIS website on the IIS server

Cmdlet	Description
Stop-IISSite	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

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

You need to identify which server roles and features have been installed on LON-SVR1. Which cmdlet would you use?

Question 2

You need to manage Hyper-V from a Windows 10 workstation. What do you need to install first to provide this capability?

Question 3

Which PowerShell module has mostly replaced the WebAdministration module for managing IIS websites?

Windows PowerShell in Windows 10

Lesson overview

In addition to network service and configuration settings, PowerShell is commonly used to configure and manage settings on a local Windows machine. PowerShell cmdlets can help to perform GUI-based operations more quickly or provide a basis to run multiple commands within a script.

In this lesson, you'll learn about common PowerShell cmdlets that can be used to perform tasks on a Windows 10 computer.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the cmdlets for managing Windows 10 devices.
- Describe the cmdlets for managing local permissions with Windows PowerShell.

Managing Windows 10 using PowerShell

You most likely use the Windows 10 GUI interface to change configuration settings and perform management tasks on Windows 10 workstations. However, you might be able to perform some tasks quicker by opening a PowerShell console and running a cmdlet.

The **Microsoft.PowerShell.Management** module includes many built-in cmdlets that can be used to obtain information and perform specific operations on a local computer. To review the cmdlets included in this module, you can enter the following:

```
Get-command -module Microsoft.PowerShell.Management
```

The following table lists some of the more common cmdlets included in the **Microsoft.PowerShell.Management** module.

Table 1: Cmdlets included in the Microsoft.PowerShell.Management module

Cmdlet	Description
Get-ComputerInfo	Retrieves all system and operating system properties from the computer
Get-Service	Retrieves a list of all services on the computer
Get-EventLog	Retrieves events and event logs from local and remote computers (only available in Windows PowerShell 5.1)
Get-Process	Retrieves a list of all active processes on a local or remote computer
Stop-Service	Stops one or more running services
Stop-Process	Stops one or more running processes
Stop-Computer	Shuts down local and remote computers
Clear-EventLog	Deletes all of the entries from the specified event logs on the local computer or on remote computers
Clear-RecycleBin	Deletes the content of a computer's recycle bin

Cmdlet	Description
Restart-Computer	Restarts the operating system on local and remote computers
Restart-Service	Stops and then starts one or more services

Running management cmdlets

The following cmdlets are examples of how to use some of the management cmdlets in Windows 10:

- To retrieve detailed information about the local computer, run the following command:

  Get-ComputerInfo

- To retrieve the latest five error entries from the Application log, run the following command:

  Get-EventLog -LogName Application -Newest 5 -EntryType Error

- To clear the Application log on the local computer, run the following command:

  Clear-EventLog -LogName Application

Additional Reading: For more information on the cmdlets available in the **Microsoft.PowerShell.Management** module, refer to **Microsoft.PowerShell.Management**[2].

Managing permissions with PowerShell

The **Microsoft.PowerShell.Security** module includes many built-in cmdlets that you can use to manage the basic security features on a Windows computer. To review the cmdlets included in this module, you can enter the following command:

```
Get-command -module Microsoft.PowerShell.Security
```

To manage access permissions on a file or folder, you use the following cmdlets included in the **Microsoft.PowerShell.Security** module.

Table 1: Cmdlets included in the Microsoft.PowerShell.Security module

Cmdlet	Description
Get-Acl	This cmdlet gets objects that represent the security descriptor of a file or resource. The security descriptor includes the access control lists (ACLs) of the resource. The ACL lists permissions that users and groups have to access the resource.
Set-Acl	This cmdlet changes the security descriptor of a specified item, such as a file, folder, or a registry key, to match the values in a security descriptor that you supply.

2 https://aka.ms/microsoft-powershell-management

Retrieving access permissions

The **Get-Acl** cmdlet displays the security descriptor for an object. For example, you can retrieve the security descriptor for a folder named **C:\Folder1**. By default, the output displays in a table format. If you pipe the output to a list format, you can review all the information included in the security descriptor.

```
Get-Acl -Path C:\Folder1|Format-List
```

By using the following command, you can retrieve a more verbose list of the access property with the file system rights, access control type, and inheritance settings for the specified object:

```
(Get-Acl -Path C:\Folder1).Access
```

You can also retrieve only specific Access properties formatted in a table format, as the following example depicts:

```
(Get-Acl -Path C:\Folder1).Access|Format-Table IdentityReference, FileSys-
temRights, AccessControlType, IsInherited
```

Updating file and folder access permissions

The **Set-Acl** cmdlet is used to apply changes to the ACL on a specific object. The process for modifying file or folder permissions consists of the following steps:

1. Use **Get-Acl** to retrieve the existing ACL rules for the object.
2. Create a new **FileSystemAccessRule** to be applied to the object.
3. Add the new rule to the existing ACL permission set.
4. Use **Set-Acl** to apply the new ACL to the existing file or folder.

The following example assigns the **Modify** permission to **C:\Folder1** for a local user named **User1**.

The first step is to declare a variable that includes the existing ACL rules for **Folder1**.

```
$ACL = Get-Acl -Path C:\Folder1
```

The second step is to create a new FileSystemAccessRule variable which specifies the access specifications to be applied:

```
$AccessRule = New-Object System.Security.AccessControl.FileSystemAccess-
Rule("User1","Modify","Allow")
```

The third step is to add the new access rule to the existing ACL rules for **Folder1**:

```
$ACL.SetAccessRule($AccessRule)
```

Finally, you need to apply the new ACL to **Folder1**:

```
$ACL | Set-Acl -Path C:\Folder1
```

Note: You can also configure an access rule to remove **Folder1** permissions for **User1** by simply changing step 3 to `$ACL.RemoveAccessRule($AccessRule).`

Copying a security descriptor to a new object

If you want to copy the exact security descriptor to a new object, you can use a combination of the **Get-Acl** and **Set-Acl** commands as follows:

```
Get-Acl -Path C:\Folder1|Set-ACL -Path C:\Folder2
```

These commands copy the values from the security descriptor of **C:\Folder1** to the security descriptor of **Folder2**. When the commands complete, the security descriptors for both folders are identical.

Test your knowledge

Use the following questions to check what you’ve learned in this lesson.

Question 1

You need to use PowerShell to review system and operating system properties of the local computer. Which cmdlet should you use?

Question 2

List the four main process steps for updating file and folder access permissions by using PowerShell.

Module 02 lab and review

Lab: Performing local system administration with PowerShell

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. You decide to complete the tasks by using Windows PowerShell.

Objectives

After completing this lab, you'll be able to:

- Create and manage Active Directory objects by using Windows PowerShell.
- Configure network settings on Windows Server by using Windows PowerShell.
- Create an Internet Information Services (IIS) website by using Windows PowerShell.

Estimated Time: 60 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

Which of the following cmdlet verbs is not associated with the `ADUser` *noun?*

☐ Get

☐ Update

☐ New

☐ Remove

☐ Set

Question 2

True or False? You can use the `New-ADObject` *cmdlet to create a new Active Directory user account.*

☐ True

☐ False

Question 3

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

- ☐ `Test-ComputerSecureChannel`
- ☐ `Get-NetIPAddress`
- ☐ `Find-NetRoute`
- ☐ `Test-Connection`

Question 4

Which PowerShell module contains the `Get-Service` *cmdlet?*

- ☐ Microsoft.PowerShell.Security
- ☐ IISAdministration
- ☐ Microsoft.PowerShell.Management
- ☐ ServerManager

Answers

Question 1

You need to use PowerShell to review the Department and email address for an Active Directory user named User1. How will you format the PowerShell command?

Get-ADUser -identity User1 -Properties Department,EmailAddress

Question 2

You need to create a Global security group named Group1. Which type of parameter will you use to specify a Global group when using the New-ADGroup PowerShell cmdlet?

You'll use the -GroupScope parameter.

Question 3

What is the default value for the -ProtectedFromAccidentalDeletion parameter of New-ADOrganizationalUnit cmdlet?

The default value is true.

Question 1

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

You can use -InterfaceAlias or -InterfaceIndex to identify a network interface when using `*-NetIPAddress` cmdlets.

Question 2

You need to manage firewall settings and rules with PowerShell. Which PowerShell module contains the required cmdlets?

The NetSecurity module contains the cmdlets to manage firewall rules.

Question 1

You need to identify which server roles and features have been installed on LON-SVR1. Which cmdlet would you use?

You would use the Get-WindowsFeature cmdlet.

Question 2

You need to manage Hyper-V from a Windows 10 workstation. What do you need to install first to provide this capability?

You need to install the Hyper-V Management tools feature on Windows 10.

Question 3

Which PowerShell module has mostly replaced the WebAdministration module for managing IIS websites?

The IISAdministration module is now used to manage common IIS website tasks.

Question 1

You need to use PowerShell to review system and operating system properties of the local computer. Which cmdlet should you use?

Get-ComputerInfo

Question 2

List the four main process steps for updating file and folder access permissions by using PowerShell.

First, use Get-Acl to retrieve the existing access control list rules for the object. Second, create a new FileSystemAccessRule to be applied to the object. Third, add the new rule to the existing ACL permission set. And finally, use Set-Acl to apply the new ACL to the existing file or folder.

Question 1

Which of the following cmdlet verbs is not associated with the `ADUser` noun?

☐ Get

■ Update

☐ New

☐ Remove

☐ Set

Explanation
The `Update` *verb isn't associated with the noun* `ADUser`*. There's no* `Update-ADUser` *cmdlet.*

Question 2

True or False? You can use the `New-ADObject` cmdlet to create a new Active Directory user account.

■ True

☐ False

Explanation
True is the correct answer. You can use either the `New-ADObject` *or* `New-ADUser` *cmdlet to create a new Active Directory user account.*

Question 3

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

☐ `Test-ComputerSecureChannel`

☐ `Get-NetIPAddress`

☐ `Find-NetRoute`

■ `Test-Connection`

Explanation
`Test-Connection` *is an alias for* `ping.exe`.

Question 4

Which PowerShell module contains the `Get-Service` cmdlet?

☐ Microsoft.PowerShell.Security

☐ IISAdministration

■ Microsoft.PowerShell.Management

☐ ServerManager

Explanation
The Microsoft.PowerShell.Management module contains the `Get-Service` *cmdlet.*

Module 3 Working with the Windows PowerShell pipeline

Understand the pipeline

Lesson overview

In this lesson, you'll learn about the Windows PowerShell pipeline and some basic techniques for running multiple commands in it. Running commands individually is both cumbersome 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'll 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.
- Review object members.
- Describe the cmdlets used to format the pipeline output for display.
- Format pipeline output.

What is the pipeline?

PowerShell can run commands in a *pipeline*, which is a chain of one or more commands in which the output from one command can pass as input to the next command. 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'll learn about those rules throughout this module.

As you interact with Windows PowerShell in the console host application, you should think of each command line as a single pipeline. You enter a command or a series of commands and then press the Enter key to run the pipeline. The output of the last command in the pipeline displays 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 enter one logical command line over multiple physical lines in the console. For example, enter **Get-Service '**, and then press the Enter key. 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. Select **Ctrl+C** to exit the command and return to the Windows PowerShell prompt.

Previously, 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 notice 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, "Filter objects out of the pipeline" explains filtering in more detail.

It helps to understand the concept of PowerShell objects and pipelines by comparing them to real-world items. For example, if we consider a car as an object, we can describe attributes of the car such as engine, car color, car size, type, make and model. In PowerShell, these would be known as properties. Properties of the object could be, in turn, objects themselves. For instance, the engine property is also an object that has attributes, such as pistons, spark plugs, crankshaft, etc.

Objects have actions, corresponding to such activities as opening or closing doors, changing gears, accelerating, and applying brakes. In PowerShell, these actions are called methods.

Pipelines allow us to take the output produced by one command and pass that object into the input of another command. To make this easier to understand we can liken commands to factories, where each factory receives materials and transforms them into something else.

Pipeline output

PowerShell commands don't 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 something that resembles a database table or a spreadsheet. In 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, PowerShell differs 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 created 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'll have to edit the second command to get the new position information. Text-based shells require significant text parsing skills. This makes scripting languages such as Perl popular, because they offer strong text parsing and text manipulation features.

In PowerShell, you instruct the cmdlet to produce the collection of service objects and to then display only the **Name** property. The structure of the objects in memory enables PowerShell to find the information for you. This way, you don't have 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, PowerShell can find the properties needed by a **Set-*** command for it to work, without you having to specify them explicitly.

Discovering object members

Members are the various components of an object and include:

- Properties, which describe attributes of the object. Examples of properties include a service name, a process ID number, and an event log message.
- Methods, which invoke an action on an object. For example, a process object can be stopped, and an event log can be cleared.
- Events, which trigger when something happens to an object. A file might trigger an event when it is updated, or a process might trigger an event when it has output to produce.

PowerShell primarily deals with properties and methods. For most commands that you run, the default on-screen output doesn't include all of an object's properties. Some objects have hundreds of properties, and the full list won't fit on the screen. PowerShell includes several configuration files that list the object properties that should display by default. That's why you notice 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 don't display on the screen by default. This command also lists methods and events and displays 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 C#.

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

To use **Get-Member**, just pipe any command output to it. For example, enter the following command in the console, and then select 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 can't use the *-WhatIf* parameter, which indicates to 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 doesn't display any output.

Demonstration: Reviewing object members

In this demonstration, you'll learn how to run commands in the pipeline and use **Get-Member**.

Demonstration Steps

Create a new global group in the IT department

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.

   Note: Note the number of returned properties and their names.
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

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 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**

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's 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 then it filters the list of properties that display and the order in which they display. Keep in mind that when you specify property names for this parameter, the original command must have returned those properties.

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 display as if the property isn't 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 displays 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 review 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, enter the following command, and then press the Enter key:

```
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*. This parameter 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*. This parameter removes the table headers from the output.
- *-Wrap*. This parameter causes text that's 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, enter the following command in the console, and then press the Enter key:

```
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 can't 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, enter the following command in the console, and then press the Enter key:

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

Demonstration: Formatting pipeline output

In this demonstration, you'll learn 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 multiple column format, and let Windows PowerShell decide the number of columns.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

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

Question 2

Describe what a pipeline is when using PowerShell.

Question 3

Which formatting cmdlet would you pipe into a PowerShell command to display the output where each property displays on a new line?

Select, sort, and measure objects

Lesson overview

In this lesson, you'll 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'll 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 PowerShell commands produce their output in a specific order. For example, the **Get-Process** and **Get-Service** commands produce output that's sorted alphabetically by name. **Get-EventLog** produces output that's sorted by time. In other cases, the output might not seem to be sorted at all. Sometimes, you might 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 sorts by the first property, then by the second property, and so on. It isn't possible in a single command to sort by one property in ascending order and another in descending order.

The following commands are examples of sorting:

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

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 review 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 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'll learn 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's sorted with the newest entry first.
6. Display a list of the 10 most recent **Security** event log entries that's sorted with the oldest entry first.
7. Display the usernames 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**.

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: 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'll frequently notice the *–Sum*, *-Minimum*, and *–Maximum* parameters truncated to *-Sum*, *-Min*, and *-Max*, corresponding to the common English abbreviations for those words. However, you can't 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 sizes:

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

Demonstration: Measuring objects

In this demonstration, you'll learn 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've already learned 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**.

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 only certain rows. **Select-Object** can select a specific number of rows. You can't use **Select-Object** to choose those rows by column or property value, only by position. You can instruct it to start from the beginning or the end of the collection. You can also instruct 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 of virtual memory use, enter the following command in the console, and then press the Enter key:

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

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

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

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

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

Selecting unique objects

Consider a scenario in which **Select-Object** evaluates 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, enter the following command in the console, and then press the Enter key:

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

Selecting object properties

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're 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 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 –Prop-
erty Name,CPU –First 10
```

Use caution when specifying property names. Sometimes, the default screen display that PowerShell creates doesn't 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 doesn't have a property that has that name. The actual property name is **CPU**. You can check 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'll learn several 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 the 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.

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 lower-case **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.

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, enter the following command in the console, and then press the Enter key:

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

Note: You can enter the command exactly as displayed and press the Enter key after the vertical pipe character. When you do so, you enter the extended prompt mode. After you enter the rest of the command, press the Enter key 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 create it a bit differently, as the following code depicts:

```
@{
 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 a string, and 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:`$PSItem` is a special variable created by Windows PowerShell. It represents whatever object was piped into the **Select-Object** command. In the previous example, that's a **Process** object. The period after `$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, such as **Name**, **ID**, and each calculated propert 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). 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 review, 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 visually suboptimal.

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 instructs 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 instructs 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's in parentheses to make sure that it runs as a single unit. You can enter this command exactly as displayed, press the Enter key on a blank line, and then review 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 with 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 learn more about the **-f** operator by running **Help About_Operators** in Windows Power-Shell.

Demonstration: Creating calculated properties

In this demonstration, you'll learn how to use **Select-Object** to create calculated properties. You'll then learn 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. Review the same list sorted by creation date, from newest to oldest.
3. Review a list of the same users in the same order but displaying the username and the age of the account, in days.

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

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

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

Question 2

You need to review all Windows services and sort them by status and then by name in descending order. How would you format the command?

Filter objects out of the pipeline

Lesson overview

In this lesson, you'll 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 a collection because it provides more flexibility. With this new technique, you'll be able to keep or remove objects based on criteria of almost any complexity.

Lesson objectives

After completing this lesson, you'll be able to:

- List the major 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 let PowerShell know 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 PowerShell comparison operators to compare a particular property of an object to a value that you specify. 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.

Table 1: Comparison operators

Operator	Description
-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 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**.

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 **-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.

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's 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**, have 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-review syntax, especially in the basic form.

For example, to display a list of only the running services, enter the following command in the console, and then press the Enter key:

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

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 PowerShell to compare. Then you specify a comparison operator, followed by the value that you want 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, PowerShell won't generate an error. Instead, your command won't 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 review. 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 can't, for example, display a list of the services that are stopped and have a start mode of **Automatic**, because that requires two comparisons.

You can't use the basic syntax with complex expressions. For example, the **Name** property of a service object consists of a string of characters. PowerShell uses a **System.String** object to contain that string of characters, and a **System.String** object has a **Length** property. The following command won't 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 older than 3.0) to reference whatever object was piped into the command. Your filter script runs one time for each object that's 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.

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's a string. Otherwise, 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's 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's 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's another common mistake:

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

The problem is that **'Starting'** isn't a complete comparison. It's 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've learned up to this point. However, when a property already contains either True or False, you don't 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's similar to reverse the logic to list only the processes that aren't 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 isn't 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 doesn't 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'll learn 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 create 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 creating 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 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're 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'll 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 }
```

However, this isn't 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's possible, check the help files for the commands that you use, to check whether they contain a parameter that can do the filtering you want.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

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*
```

Question 2

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

Module 03 lab A and review

Lab A: Using PowerShell pipeline

Scenario

One of your administrative tasks at Adatum Corporation is to configure advanced PowerShell scripts. You need to ensure that you understand the foundations of working with PowerShell pipeline by sorting, filtering, enumerating, and converting objects.

Objectives

After completing this lab, you'll be able to:

- Select, sort, and display data by using pipelines.
- Filter objects out of the pipeline by using basic and advanced syntax forms.

Estimated Time: 60 minutes

Module review

Use the following questions to check what you've learned in the lessons of this module.

Question 1

True or False? The `Format-Wide` *cmdlet accepts the* `-AutoSize` *and* `-Wrap` *parameters.*

☐ True

☐ False

Question 2

True or False? The `ConvertTo-Html` *cmdlet can use a Cascading Style Sheet (CSS) to offer specific visual styles.*

☐ True

☐ False

Enumerate objects in the pipeline

Lesson overview

In this lesson, you'll 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's a building block for creating automation scripts.

Lesson objectives

After completing this lesson, you'll 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, PowerShell doesn't 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, consider an object produced when you run **Get-ChildItem -File** to list files on a disk drive. The resulting 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 into Windows PowerShell, so, if you want to run that method on many file objects, enumeration allows you to accomplish this with a single 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's an example:

```
Get-ChildItem –Path C:\Encrypted\  File | ForEach Object  MemberName En-
crypt
```

With this syntax, you don't include the parentheses after the member name if the member is a method. Because this basic syntax is meant to be short, you'll frequently notice it without the *-MemberName* parameter name, and you might notice it 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 run into many scenarios that require enumeration. Each new operating system and version of PowerShell introduces new PowerShell commands. Newer operating systems typically introduce new commands which perform actions that previously required enumeration.

Limitations of the basic syntax

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

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

Demonstration: Basic enumeration

In this demonstration, you'll learn 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 one command, or it can include many commands in sequence.

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

```
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 doesn't have any input arguments. For methods that do need input arguments, provide them as a comma-separated list inside the parentheses. Don't 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` display in the script block, the actual integers aren't 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'll learn 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.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

If you have programming or scripting experience, is ForEach-Object familiar to you?

Question 2

Do you prefer the basic or advanced syntax of ForEach-Object?

Send pipeline data as output

Lesson overview

When you provide information about your network infrastructure, it's 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'll learn how to send pipeline data to files and in various output formats.

Lesson objectives

After completing this lesson, you'll 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.
- Describe how PowerShell matches incoming pipeline data to the parameters of a cmdlet.

Writing output to a file

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 display on the screen.

PowerShell also supports converting and exporting objects, which the next topics examine. The **Out-File** behavior differs from converting or exporting the objects, because you don’t change the form of the objects. Instead, PowerShell captures what would have displayed 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. 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 PowerShell to external storage. However, the text files that **Out-File** creates are usually intended for reviewing by a person. Therefore, it's frequently difficult or impractical to read the data back into Windows PowerShell in any way that allows the data to be manipu-lated, sorted, selected, and measured.

Out-File doesn't produce any output of its own. This means that the command doesn't put objects into the pipeline. After you run the command, you should expect no output on the screen.

Keeping track of what the pipeline contains

As you begin to create more complex commands in PowerShell, you need to become accustomed to keeping track of the pipeline's contents. As each command in the pipeline runs, it might produce output different from the input it received. Therefore, the next command will work with something different.

For example, review the following:

```
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 displays 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 doesn't run correctly the first time. The best way to debug is to start with one command and check what it produces. Then add the second command, and check 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 seem to be correctly sorted, because the sorting was done on the **Name** property and not the **DisplayName** property.

Converting output to CSV

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 comma-separated value (CSV) format. Converting to CSV is useful for reviewing and manipulating large amounts of data, because you can easily open the resulting file in a program such as Microsoft Excel.

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 PowerShell. You can pipe the data to another command that writes the data to a file or manipulates it in another way.

Here's an example:

```
Get-Service | ConvertTo-Csv | Out-File Services.csv
```

A command that uses **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's an example:

```
Get-Service | Export-Csv Services.csv
```

Export commands, such as **Export-Csv**, combine the functionality of **ConvertTo** with a command such as **Out-File**. Export commands don't usually put any output into the pipeline. Therefore, nothing displays 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's generally more difficult to manipulate within Windows PowerShell. For example, you can't easily sort, select, or measure data that's been converted.

As noted previously, 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 with the **Import-Csv** command. **Import-Csv** creates objects that have properties matching the columns in the CSV file.

Converting output to XML

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 don't 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.

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 doesn't 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 JavaScript Object Notation (JSON) format. JSON is very popular in web application development because of its compact size and flexibility. As the name might suggest, it's very easy for JavaScript to process.

The JSON format represents data in name-value pairs that resemble and work like hash tables. So, like a hash table, JSON doesn't consider what kind of data exists in the name or value. It's up to the script or application consuming the JSON data to understand what kind of data is represented.

In 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 doesn't have an **Export** command for 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.

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, which 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's coded as HTML. You can control the HTML format in a limited way 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 display before the table or list output.
- *-PostContent*. Defines any content that should display after the table or list output.

Demonstration: Exporting data

In this demonstration, you'll learn 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

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 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 PowerShell to prompt you for input after it displays one page of data on the screen.

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 review, sort, filter, and analyze output, and you don't 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 a Microsoft 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 can't do is save the data directly from the grid view window.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

What other data formats might you want to convert data to or from?

Question 2

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Module 03 lab B and review

Lab B: Using PowerShell pipeline

Scenario

One of your administrative tasks at Adatum Corporation is to configure advanced PowerShell scripts. You need to ensure that you understand the foundations of working with PowerShell pipeline by sorting, filtering, enumerating, and converting objects.

Objectives

After completing this lab, you'll be able to:

- Enumerate pipeline objects by using basic and advanced syntax.
- Convert objects to different formats.

Estimated Time: 60 minutes

Module review

Use the following questions to check what you've learned in the lessons of this module.

Question 1

True or False? `$PSItem` *and* `$_` *are functionally the same and will present similar results when used in a PowerShell script.*

☐ True

☐ False

Question 2

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Pass pipeline objects

Lesson overview

In this lesson, 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.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe pipeline parameter binding.
- Identify ByValue parameters.
- Pass data by using ByValue.
- Identify ByPropertyName parameters.
- Pass data ByPropertyName.
- Pass pipeline data ByPropertyName.
- Use manual parameters to override the pipeline.
- Use parenthetical commands.
- Expand property values.

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
```

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, which is 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.

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 **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
```

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           Definition
----             ----------           ----------
```

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'll learn 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 data by using 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.

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
```

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 by using ByPropertyName

In this demonstration, you'll learn 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**.

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 the Enter key:

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

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'})
```

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.

Demonstration: Using parenthetical commands

In this demonstration, you'll learn 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 the Enter key:

```
Get-ADGroup "London Users" | Add-ADGroupMember
```

Note that a prompt to enter users appears. Press the Enter key. You will see an error because no member was specified.

4. In the console, type the following command, and then press the Enter key:

```
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.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Question 2

What are some reasons you would use parenthetical commands instead of the pipeline?

Answers

Question 1

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

It depends on who wrote the command that produced the object. For most Microsoft commands, the MSDN Library documents the output objects. Using the object's type name in an internet search is frequently the fastest way to find existing documentation about an object.

Question 2

Describe what a pipeline is when using PowerShell.

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.

Question 3

Which formatting cmdlet would you pipe into a PowerShell command to display the output where each property displays on a new line?

The Format-List cmdlet formats the output of a command where each property displays on a new line.

Question 1

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

When you don't need the entire output of a command, selecting only the first rows can improve performance. Select-Object can indicate to the previous commands that it requires no more data. Some commands can then stop processing, thereby reducing the total amount of time it takes Windows PowerShell to complete the task.

Question 2

You need to review all Windows services and sort them by status and then by name in descending order. How would you format the command?

get-service|sort status,name -descending

Question 1

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*
```

No. The following command offers a better approach: Get-Service –Name svc.*
In this case, it filters the names as part of the Get-Service command.

Question 2

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

This is a personal opinion. The $PSItem variable is new in Windows PowerShell 3.0, so experienced users frequently use $_ out of habit. The $_ variable is shorter and easier to enter, but for many beginners, it's more visually confusing than $PSItem. Both work the same way, and you'll probably notice both in various online examples, books, and other resources. Many of the examples in the Windows PowerShell help files still use $_.

Question 1

True or False? The `Format-Wide` cmdlet accepts the `-AutoSize` and `-Wrap` parameters.

☐ True

■ False

Explanation
False is the correct answer. The `Format-Wide` *cmdlet doesn't accept the* `-Wrap` *parameter.*

Question 2

True or False? The `ConvertTo-Html` cmdlet can use a Cascading Style Sheet (CSS) to offer specific visual styles.

■ True

☐ False

Explanation
Yes. You can either embed a style sheet by providing the appropriate HTML and CSS code to the `-Head` *parameter or attach an external style sheet by using the* `-CssUri` *parameter.*

Question 1

If you have programming or scripting experience, is ForEach-Object familiar to you?

The command is functionally similar to the enumeration programming constructs that many programming and scripting languages have. For example, in Microsoft Visual Basic, the ForEach construct provides a similar purpose. PowerShell does have a ForEach scripting construct, although its syntax differs from that of the ForEach-Object command.

Question 2

Do you prefer the basic or advanced syntax of ForEach-Object?

Answers will vary. However, remember that you'll probably encounter both forms of the syntax as you discover examples, such as in books or blogs, that other PowerShell users have created.

Question 1

What other data formats might you want to convert data to or from?

Many formats exist. The .xls and the .xlsx formats are common requests, although PowerShell doesn't contain a native command for reading or writing those formats.

Question 2

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Commands that use the noun Object work with any kind of input. Therefore, each of them defines a parameter named –InputObject. That parameter receives any kind of object from the pipeline, and therefore, it accepts input of the type Object or PSObject from the pipeline by using ByValue.

Question 1

True or False? `$PSItem` and `$_` are functionally the same and will present similar results when used in a PowerShell script.

[✓ True]

False

Explanation
True is the correct answer. Both are functionally the same. Windows PowerShell 3.0 introduced `$PSItem` *as an easier-to-review alternative to* `$_`*, so* `$PSItem` *isn't available in earlier versions of Windows PowerShell. Therefore, scripts that have to maintain backward compatibility must continue to use* `$_`*. You're likely to notice* `$_` *in examples, such as in online articles or blogs created by other people for earlier versions, so you should remember both* `$_` *and* `$PSItem`*.*

Question 2

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Commands that use the noun Object work with any kind of input. Therefore, each of them defines a parameter named –InputObject. That parameter receives any kind of object from the pipeline, and therefore, it accepts input of the type Object or PSObject from the pipeline by using ByValue.

Question 1

Why do most commands that use the noun Object have an –InputObject parameter that accepts objects of the type Object or PSObject?

Commands that use the noun Object work with any kind of input. Therefore, each of them defines a parameter named –InputObject. That parameter receives any kind of object from the pipeline, and therefore, it accepts input of the type Object or PSObject from the pipeline by using ByValue.

Question 2

What are some reasons you would use parenthetical commands instead of the pipeline?

Several answers are possible. Some of them include:

Module 4 Using PSProviders and PSDrives

Using PSProviders

Lesson overview

In this lesson, you'll learn about PSProviders. *PSProviders* are adapters that connect Windows PowerShell to data stores. They offer an easier-to-understand and consistent interface for working with data stores. You can also reuse scripts as you change the underlying technologies with which you're working.

Lesson objectives

After completing this lesson, you'll be able to:

- Explain the purpose of PSProviders.
- Compare different PSProvider capabilities.
- Explain how to access PSProvider help files.
- Explain how to review a list of providers and the help options for a specific provider.

What are Windows PowerShell providers?

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 managing hard drives using command-line commands, PSProviders help those administrators manage other forms of data storage using the same familiar commands.

A provider presents data as a hierarchical store. For example, 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 their 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 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's dynamic, which makes it suitable for technologies that are subject to frequent changes. For example, when managing IIS, its provider can accommodate newly introduced Microsoft and third-party IIS add-ins. Even though using a provider to manage dynamic and extensible technologies tends to be more complex, it offers a more consistent approach due to its extensibility.

Some common providers include:

- Registry. Provides access to the registry keys and values.
- Alias. Provides access to aliases for Windows PowerShell cmdlets.
- Environment. Provides access to Windows environment variables and their values.
- FileSystem. Provides access to the files and folders in the file system.
- Function. Provides access to Windows PowerShell functions loaded into memory.
- Variable. Provides access to Windows PowerShell variables and their values loaded into memory.

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'll receive an error message. You'll also receive an error message whenever you use a common parameter that the provider doesn't support.

Running the **Get-PSProvider** cmdlet lists 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** for providers that can support the *–WhatIf* and *–Confirm* parameters.
- **Filter** for providers that support filtering.
- **Include** for providers that can include items in the data store based on the name. Supports using wildcards.
- **Exclude** for providers that can exclude items in the data store based on the name. Supports using wildcards.
- **ExpandWildcards** for providers that support wildcards in their paths.
- **Credentials** for providers that support alternative credentials.
- **Transactions** for providers that support transacted operations.

You should always review the capabilities of a provider before you work with it. This helps you avoid unexpected errors when you try to use unsupported capabilities.

Accessing provider help

You can display a list of available providers by using the **Get-PSProvider** cmdlet. Be aware that providers can be added into Windows PowerShell when you load modules, and they don't display until loaded. For example, when you run the **Import-Module ActiveDirectory** command to load the **ActiveDirectory** module or use module autoloading when running an Active Directory cmdlet, a PSProvider for Active Directory is included.

Some providers include help files that you can review. Help files use the naming format **about_Provider-Name_Provider**. For example, the help file for the FileSystem provider is **about_FileSystem_Provider**. You can review this help file's contents by running the following command:

```
Get-Help about_FileSystem_Provider
```

Cmdlets that work with providers use the nouns **Item** and **ItemProperty**. To list cmdlets that work with providers, run the following cmdlets:

```
Get-Command *-Item,*-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: Reviewing PSProvider help

In this demonstration, you'll learn how to review a list of providers and help for a provider.

Demonstration steps

1. Use **Get-PSProvider** to display the list of loaded providers. Notice the capabilities listed for each one.
2. Use **Import-Module** to load the **ActiveDirectory** module.
3. Use **Get-PSProvider** to display the list of providers again. Notice the new provider that the **ActiveDirectory** module added.
4. Use **Get-Help** to display help topics related to the term **Registry**.
5. Display help for the **about_Registry_Provider** topic.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which of the following options are PSProviders that load automatically when you open a Windows PowerShell prompt? (Choose three.)

☐ Registry

☐ HKCU

☐ Alias

☐ FileSystem

☐ ActiveDirectory

Question 2

Which command would you use to get help information for the Registry provider?

- ☐ `Get-Command Registry`
- ☐ `Get-Help about_Registry_Provider`
- ☐ `Get-Command Registry_Provider`
- ☐ `Get-Help Registry`
- ☐ `Get-Help Registry_Provider`

Using PSDrives

Lesson overview

In this lesson, you'll learn how to work with PSDrives. A *PSDrive* represents a specific form of storage that connects to Windows PowerShell by using a PSProvider. To use PSProviders successfully, you need to understand how to work with PSDrives. In some cases, PSDrives are the primary interface for working with the underlying data.

Lesson objectives

After completing this lesson, you'll 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 it uses to make the connection.

Names identify drives in Windows PowerShell. Drives can consist of a single letter. Single-letter drive names typically connect to a **FileSystem** drive. For example, drive **C** connects to the physical drive **C** of a computer. However, names also can consist of more than one character. For example, the drive **HKCU** connects to the **HKEY_CURRENT_USER** registry hive.

To create a new connection, you 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 PSProvider's capabilities, you might also specify alternative credentials and other options.

Windows PowerShell always starts a new session with the following drives:

- Registry drives **HKLM** and **HKCU**
- Local hard drives, such as drive **C**
- Windows PowerShell storage drives **Variable**, **Function**, and **Alias**
- Web Services for Management (WS-Management) settings drive **WSMan**
- Environment variables drive **Env**
- Certificate store drive **CERT**

You can review a list of drives by running the **Get-PSDrive** cmdlet.

Note: Drive names don't 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**. Cmdlets such as **New-PSDrive** require a drive name, but when using these commands, don't include a colon in the drive name.

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 associated with PSDrives without realizing it. PSDrives contain items that 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** cmdlet with the *-Noun* parameter to review a list of commands that work on each PSDrive object. You can also use **Get-Help** to review the help for each command. The following table describes the verbs that are associated with common **PSDrive** cmdlets.

Table 1: PSDrive cmdlet verbs

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 or 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.
Move	Moves an item or item property from one location to another.
Remove	Deletes an item or item property.
Rename	Renames an item or item property.
Invoke	Performs the default action that's associated with an item.

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 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 doesn't allow item names to contain the asterisk or question mark characters.
- *–LiteralPath*. This parameter treats all characters as literals and doesn't interpret any character as a wildcard. The literal path ***.txt** *means the item named* ".txt." This approach is useful in drives where the asterisk and question mark 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 those described in the following table.

Table 2: Location commands

Command	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 Windows Commamd Prompt (**cmd.exe**) console. In PowerShell, **pushd** and **popd** are aliases for those cmdlets.

Additional reading: For more information about location stacks, refer to **Push-Location**[1].

Working with the file system

Administrators who are familiar with using the Windows Command Prompt (**cmd.exe**) 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.

You can use the **Get-Alias** or **Get-Command** cmdlets to identify the cmdlets that map to these aliases and functions. Keep in mind that the aliases and functions aren't exact duplicates of the original **cmd.exe** commands, but instead the syntax of an alias matches the corresponding cmdlet. For example, **Dir** is an alias for the **Get-ChildItem** cmdlet. 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** as you would when using Windows Command Prompt.

Note: Because Windows PowerShell accepts a slash (/) or backslash (\) as a path separator, Windows PowerShell interprets **Dir /s** to display a directory listing for the folder named **s**. If a folder named **s** exists, the command appears to work and doesn't display an error. If no such folder exists, it displays an error.

Moving within the file system

You can move within the file system by using the **Set-Location** cmdlet. This cmdlet functions similar to the Windows Command Prompt command **Cd**. When using it, you can specify either an absolute or a relative path. For example, **Set-Location C:\Users** changes to the C:\Users folder. **Set-Location Temp** changes to the Temp folder that's one level down from the current directory.

Create new files or folders

You can create new files and folders by using the **New-Item** cmdlet. You include the *-Path* parameter to define the name and location, and the *-ItemType* parameter to specify whether you want to create a file or directory.

1 https://aka.ms/idat4p

Delete files or folders

You can remove files or folders with the **Remove-Item** cmdlet 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. Otherwise, or you'll be asked to confirm the action.

Find and enumerate files or folders

Use the **Get-Item** cmdlet 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 returns all files and folders in the current directory. The **Get-Item *** command is equivalent to the **Get-ChildItem** cmdlet, which returns all the children of a specified path.

You can use the **Get-ChildItem** cmdlet with the *-Recurse* switch to enumerate through child files and folders. The FileSystem provider also supports the *-Exclude*, *-Include*, and *-Filter* parameters. These 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'll learn how to manage the file system by using Windows PowerShell.

Demonstration steps

1. Use the **Cd** alias to change location to C:.
2. Use the **Set-Location** cmdlet to go to **C:\Windows**.
3. Create a new PSDrive named **WINDIR** that maps to the **C:\Windows** folder.
4. Display a directory listing of the **WINDIR** drive by using the **Dir** alias.
5. Display the directory listing for the **WINDIR** drive by using the **Get-ChildItem** cmdlet.
6. Use the **New-Item** cmdlet to create a folder named **Temp** in the **E:\Mod04** folder.

Working with the registry

Experienced system administrators are familiar with the graphical Registry Editor, which they can use to manage registry keys, entries, and values. However, you can also manage the registry by using Windows PowerShell and the Registry provider.

You can use the **New-PSDrive** cmdlet to create PSDrives for any part of the registry. PowerShell uses the Registry provider to create two PSDrives automatically:

- **HKLM**. Represents the **HKEY_LOCAL_MACHINE** registry hive.
- **HKCU**. Represents the **HKEY_LOCAL_USER** registry hive.

You access registry keys by using cmdlets with the **Item** and **ChildItem** nouns, whereas you access entries and values by using cmdlets with the **ItemProperty** and **ItemPropertyValue** nouns. This is because PowerShell considers registry entries to be properties of a key item.

To return all the registry keys under the **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion** path, run the following command:

```
Get-ChildItem HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Within the registry, a registry key is equivalent to a folder within a file system that's used to organize information. The information used by apps is stored in registry values. The value name is a unique identifier for the value, and the value data is the information used by apps.

For example, to return the registry values under the **HKLM\SOFTWARE\Microsoft\Windows\Current-Version\Run** path, run the following command:

```
Get-ItemProperty HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

You can use the **Get-ItemPropertyValue** cmdlet to obtain the value of a specific registry entry. For example, if you want to return the path to the Windows Defender executable identified by the value **WindowsDefender** entry, run the following command:

```
Get-ItemPropertyValue HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
-Name WindowsDefender
```

Note: The Registry provider doesn't support the **Invoke-Item** cmdlet. There's no default action for registry keys, entries, or values.

The Registry provider supports a dynamic parameter, -*Type*, for the ***-ItemProperty** cmdlets that are unique to the Registry provider. The following table lists valid parameter values and their equivalent registry data types.

Table 1: 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 together without worrying that some of the settings will be updated successfully while others might 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: Remember to back up your registry settings before you attempt to modify registry keys and values. You can export registry settings to a file by using the **reg.exe** command.

Demonstration: Managing the registry

In this demonstration, you'll learn how to manage the registry by using Windows PowerShell.

Demonstration steps

1. Use the **Set-Location** cmdlet to set the working folder location to **HKLM:\Software**.

2. Use the **Get-ChildItem** cmdlet to review the registry keys under **HKLM:\Software**.
3. Use the **New-Item** cmdlet to create a registry key named **Demo** in the **HKLM:\Software** path.
4. Use the **Add-ItemProperty** cmdlet to add a value to the **Demo** key:

    - Name: **Version**
    - Value: **1.0.0**
    - Type: **String**

5. Use the **Get-ItemProperty** cmdlet to review the registry values for the **Demo** key.

Working with certificates

If you've reviewed or managed security certificates on a client or server computer, you've probably used the Certificates snap-in for the Microsoft Management Console (MMC). The Certificates snap-in enables you to browse the certificates stores on local or remote computers. The Windows PowerShell Certificates provider also allows you to review 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 the users and the local computer. These locations are **Curren-tUser** and **LocalMachine**. The certificates specific to the user or computer are in the **My** subfolder, represented by the **Cert:\CurrentUser\My** notation.

The Certificates provider supports the **Get**, **Set**, **Move**, **New**, **Remove**, and **Invoke** verbs in combination with the **Item** and **ChildItem** nouns. (Note that the **ItemProperty** noun is not supported.) All ***-Location** commands are also supported.

The **Invoke-Item** cmdlet in combination with the Certificates provider opens the MMC with the Certificates snap-in automatically loaded.

The **Get-ChildItem** command has a variety of dynamic parameters that are unique to the **Certificate** provider. These include:

- *-CodeSigningCert*. Gets certificates that can be used for code signing.
- *-DocumentEncryptionCert*. Gets certificates for document encryption.
- *-DnsName*. Gets certificates with the domain name in the **DNSNameList** property of the certificate. This parameter accepts wildcards.
- *-EKU*. Gets certificates with the specified text in the **EnhancedKeyUsageList** property. This parameter supports wildcards.
- *-ExpiringInDays*. Gets certificates that are expiring within the specified number of days.
- *-SSLServerAuthentication*. Gets only Secure Sockets Layer (SSL) server certificates.

There are also many certificate management cmdlets in the **pki** module that don't require you to use the **Cert** drive. For example, to create a self-signed certificate for the server `webapp.contoso.com`, use the following code:

```
New-SelfSignedCertificate -DnsName "webapp.contoso.com" -CertStoreLocation
"Cert:\LocalMachine\My"
```

Note: For a list of certificate management cmdlets in the **pki** module, run **Get-Command -Module pki**.

Working with other PSDrives

In addition to the file system drives, the registry drives, and the **Cert** drive, Windows PowerShell includes other drives:

- **Alias**. Review and manage Windows PowerShell aliases.
- **Env**. Review and manage Windows environment variables.
- **Function**. Review and manage Windows PowerShell functions.
- **Variable**. Review and manage Windows PowerShell variables.
- **WSMan**. Review and manage WS-Management configurations.

There are a number of providers included with other modules that can create PSDrives. For example, the process of installing management tools for Windows Server roles often includes additional drives such as:

- **AD**. This drive is created by the ActiveDirectory provider, which is part of the ActiveDirectory module included with the Remote Server Administration Tools (RSAT). The ActiveDirectory provider supports reviewing and managing AD DS database contents, such as user and computer accounts.
- **IIS**. This drive is created by the WebAdministration provider, which is part of the WebAdministration module that's included with IIS management tools. The WebAdministration provider allows you to review and manage application pools, websites, web applications, and virtual directories.

Note: The ActiveDirectory module includes many cmdlets for managing Active Directory objects. To review the cmdlets in the ActiveDirectory module, run **Get-Command -Module ActiveDirectory**.

Note: The WebAdministration module includes many cmdlets for managing IIS. To review the cmdlets in the WebAdministration module, run **Get-Command -Module WebAdministration**.

These additional drives support using most of the standard provider verbs and nouns. There might also be specific cmdlets that can perform the same functions. For instance, you can use the **Get-Alias** cmdlet or the following provider-based command to return a list of all aliases in the current Windows PowerShell session:

```
Get-Item -Path Alias:
```

In some cases, there are no equivalent cmdlets for a provider-based command. For example, there's no **Remove-Alias** cmdlet that deletes an alias, but you can use either of the following commands to delete an alias named **MyAlias**:

```
Remove-Item -Path Alias:MyAlias
```

```
Clear-Item -Path Alias:MyAlias
```

As with the other providers covered by earlier topics, the providers used to create these drives 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.

To understand the details of what you can do with an item that's accessible through a drive, you should review the help for the provider that's used to create the drive. In the help, you can identify any dynamic parameters or properties. You can identify the provider used to create a drive by using the **Get-PSDrive** cmdlet. The you can use the **Get-Help** cmdlet to review the help available for the provider. For example, you could use the command to review help for the Alias provider:

```
Get-Help About_Alias_Provider
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which PSDrives are created by the Registry provider? (Choose two.)

☐ Registry

☐ HKCU

☐ AD

☐ HKLM

☐ C

Question 2

True or False? You can use the FileSystem provider to create PSDrives that are mapped to a network share.

☐ True

☐ False

Module 04 lab and review

Lab: Using PSProviders and PSDrives with PowerShell

Scenario

You're a system administrator for the London branch office of Adatum Corporation. You must reconfigure several settings in your environment. You've recently learned about PSProviders and PSDrives, and that you can use them to access data stores. You've decided to use PSDrives to reconfigure these settings.

Objectives

After completing this lab, you'll be able to:

- Use a PSDrive to create files and folders.
- Create registry keys and values.
- Use a PSDrive to create and view Active Directory objects.

Estimated time: 30 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

True or false? To create a PSDrive, you need to specify the PSProvider that will be used.

☐ True

☐ False

Question 2

Which cmdlet should you use to determine the capabilities of a PSProvider?

☐ `Get-ChildItem`

☐ `Get-Item`

☐ `Get-PSProvider`

☐ `Get-PSDrive`

Question 3

Which cmdlet should you use to create a registry key value?

- ☐ `New-Item`
- ☐ `New-ItemProperty`
- ☐ `New-ChildItem`
- ☐ `New-PSDrive`

Question 4

Which certificate stores are available through the Cert PSDrive? (Select two.)

- ☐ CurrentUser
- ☐ HKCU
- ☐ LocalMachine
- ☐ FileSystem
- ☐ HKLM

Question 5

Which PSDrive is created automatically when the WebAdministration provider is loaded?

- ☐ AD
- ☐ HKLM
- ☐ FileSystem
- ☐ IIS

Answers

Question 1

Which of the following options are PSProviders that load automatically when you open a Windows PowerShell prompt? (Choose three.)

- ■ Registry
- ☐ HKCU
- ■ Alias
- ■ FileSystem
- ☐ ActiveDirectory

Explanation
The Registry, Alias, and FileSystem providers are loaded automatically when you open a Windows PowerShell prompt. HKCU is a PSDrive, not a provider. The ActiveDirectory provider is loaded only after you load the ActiveDirectory module.

Question 2

Which command would you use to get help information for the Registry provider?

- ☐ `Get-Command Registry`
- ■ `Get-Help about_Registry_Provider`
- ☐ `Get-Command Registry_Provider`
- ☐ `Get-Help Registry`
- ☐ `Get-Help Registry_Provider`

Explanation
`Get-Help about_Registry_Provider` *is the correct answer. Help topics for providers have names in the format of* `about_ProviderName_Provider`.

Question 1

Which PSDrives are created by the Registry provider? (Choose two.)

- ☐ Registry
- ■ HKCU
- ☐ AD
- ■ HKLM
- ☐ C

Explanation
HKCU and HKLM are the correct answers. These PSDrives are for the HKEY_Current_User and HKEY_Local_Machine hives of the registry.

Question 2

True or False? You can use the FileSystem provider to create PSDrives that are mapped to a network share.

■ True

☐ False

Explanation
True is the correct answer. The FileSystem provider can create PSDrives that are mapped to folders in the local filesystem or a network share.

Question 1

True or false? To create a PSDrive, you need to specify the PSProvider that will be used.

■ True

☐ False

Explanation
True is the correct answer. If you don't specify a provider when you run the `New-PSDrive` *cmdlet, you're prompted for a provider.*

Question 2

Which cmdlet should you use to determine the capabilities of a PSProvider?

☐ `Get-ChildItem`

☐ `Get-Item`

■ `Get-PSProvider`

☐ `Get-PSDrive`

Explanation
`Get-PSProvider` *is the correct answer. The list of providers that* `Get-PSProvider` *returns includes each of the provider's capabilities.*

Question 3

Which cmdlet should you use to create a registry key value?

☐ `New-Item`

■ `New-ItemProperty`

☐ `New-ChildItem`

☐ `New-PSDrive`

Explanation
`New-ItemProperty` *is the correct answer. Registry key values are considered a property of a registry key.*

Question 4

Which certificate stores are available through the Cert PSDrive? (Select two.)

■ CurrentUser

☐ HKCU

■ LocalMachine

☐ FileSystem

☐ HKLM

Explanation
The correct answers are CurrentUser and LocalMachine. These are the top-level folders in the Cert PSDrive.

Question 5

Which PSDrive is created automatically when the WebAdministration provider is loaded?

☐ AD

☐ HKLM

☐ FileSystem

■ IIS

Explanation
IIS is the correct answer. The WebAdministration provider creates the IIS drive to provide access to manage websites.

Module 5 Querying management information by using CIM and WMI

Understand CIM and WMI

Lesson overview

In this lesson, you'll learn about the architecture of Common Information Model (CIM) and Windows Management Instrumentation (WMI). Both technologies connect to a common information repository that contains 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. An understanding of the framework and syntax of CIM and WMI will help you know and control almost every aspect of an operating system environment.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the architecture of CIM and WMI.
- Explain the purpose of the repository.
- Explain how to locate online documentation for repository classes.
- Locate online class documentation.

Architecture and technologies

Windows Management Instrumentation (WMI) and Common Information Model (CIM) are related technologies. Both technologies are based on industry standards that the Distributed Management Task Force (DMTF) defines. The DMTF 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 (WBEM) standard, is an older technology that's based on preliminary standards and proprietary technology. CIM is a newer technology that's based on open, cross-platform standards.

Both technologies provide a way to connect to a common information repository (also known as the *WMI repository*). This repository contains management information that you can query and manipulate. There are two parallel sets of cmdlets that you can use to perform tasks by using WMI or CIM.

CIM cmdlets

You can use CIM cmdlets to access the repository on local or remote computers. Multiple protocols for connectivity are supported and the protocol varies, depending on how the connection is created. Component Object Model (COM) is a protocol for local communication between software components. Distributed COM (DCOM) is an older proprietary Microsoft protocol for connectivity which uses dynamic ports. Web Services for Management (WS-MAN) is a web-based protocol defined by the DMTF for connectivity. A WS-MAN listener is configured automatically on port **5985**. Also, a secure listener is created on port **5986** if a certificate is available.

CIM cmdlets can create connections in three ways:

- Local connections. When you don't specify a remote computer, the CIM cmdlets use a local COM session to access and communicate with the repository.
- Ad-hoc connections. When you specify a remote computer by using the *-ComputerName* parameter with a CIM cmdlet, the remote connection uses WS-MAN.
- CIM sessions. You can precreate a CIM session with specific options for connectivity to remote computers. When you create a CIM session, you can specify whether to use WS-MAN or DCOM.

To use the CIM cmdlets on a remote computer, the remote computer doesn't need to have PowerShell installed. The remote computer needs to support WS-MAN to connect to it with CIM cmdlets. You can use CIM cmdlets with Linux computers.

In Windows, WS-MAN connectivity is provided by the Windows Remote Management (WinRM) service. This service also provides support for Windows PowerShell remoting, but PowerShell remoting is not required for the CIM cmdlets.

WMI cmdlets

WMI cmdlets use the same repository as CIM cmdlets. The only difference is how the WMI cmdlets connect to a remote computer.

WMI cmdlets don't support session-based connections. These commands support only ad-hoc remote 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, which connects by using randomly selected port numbers. Special firewall exceptions are required for DCOM to work correctly.

The Windows Management Instrumentation service provides the connectivity for WMI. WMI cmdlets don't require any version of the Windows Management Framework on a remote computer. They also don't require Windows PowerShell remoting to be enabled. If the remote computer has the Windows Firewall feature enabled, you need to ensure that the remote administration exceptions are 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?

In general, you should use CIM cmdlets instead of the older WMI cmdlets. 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 aren't practical.

CIM cmdlets also provide easier network connectivity. Because CIM cmdlets use WS-MAN for remote connectivity, the firewall rules for known ports are easier to create than the randomly generated ports used by DCOM. If remote computers don't support using WS-MAN, you can specify the use of DCOM for connectivity by using a CIM session.

Using cmdlets 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. If there is a PowerShell cmdlet for managing a specific item such as networking, then it's typically easier to use that cmdlet instead of the equivalent class methods exposed through CIM cmdlets. These purpose-built cmdlets often use CIM or WMI but hide their complexity from you. This approach gives you the advantages of Windows PowerShell cmdlets, such as discoverability and built-in documentation, while also giving you the existing functionality of the repository.

Understanding the repository

The repository that Common Information Model (CIM) and Windows Management Instrumentation (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 doesn't exist on other computers.

To find the top-level namespaces, run the following command:

```
Get-CimInstance -Namespace root -ClassName __Namespace
```

Note: In the previous command, there are two underscores (_) in the class name of `__Namespace`.

Some of the namespaces returned might include:

- subscription
- DEFAULT
- CIMV2
- msdtc
- Cli
- Intel_ME
- SECURITY
- HyperVCluster
- SecurityCenter2
- RSOP
- Intel

- PEH
- StandardCimv2
- WMI
- directory
- Policy
- virtualization
- Interop
- Hardware
- ServiceModel
- SecurityCenter
- Microsoft
- Appv
- dcim

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'll have two instances of the class that represents processors. If your computer doesn't have an attached tape drive, you'll have zero instances of the tape drive class.

Instances are objects, similar to the objects that you've 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

Although many Microsoft product groups and independent software vendors expose management information in the repository, only some of them offer official documentation. In most cases, an internet search for a class name provides your best option for finding the documentation that exists.

Classes provided by the Win32 provider are commonly used and well-documented. These classes are accessible through Common Information Model (CIM) and Windows Management Instrumentation (WMI) cmdlets. The names for these classes use the format Win32_*ObjectType*. For example, you can access operating system information by using the **Win32_OperatingSystem** class.

Additional reading: You can review the documentation for Win32 classes on **Win32 Provider**[1].

CIM classes are also well-documented and they are available when using the CIM cmdlets. The names for these classes use the format CIM_*ObjectType*. For example, you can query operating system information by using the **CIM_OperatingSystem** class. Some CIM classes provide the same information as an equivalent Win32 class.

Additional reading: You can review the documentation for the CIM classes on **CIM Classes (WMI)**[2].

To a certain extent, classes are self-documenting. You can use the **Get-Member** cmdlet to review the properties and methods available for a class. To do this, you query an instance of a class and then pipe it

1 https://aka.ms/Win32-provider
2 https://aka.ms/CIM-classes-wmi

to **Get-Member**. For example, to get properties and methods for the **Win32_OperatingSystem** class, run the following command:

```
Get-CimInstance -Namespace root\cimv2 -ClassName Win32_OperatingSystem |
Get-Member
```

Windows PowerShell Help doesn't contain any information about CIM and WMI classes. CIM and WMI are external technologies that Windows PowerShell can use and understand. However, because they are external technologies, Windows PowerShell Help doesn't document the repository classes.

Demonstration: Finding documentation for classes

In this demonstration, you'll learn how to locate online class documentation.

Demonstration steps

- Using the host computer, use a search engine to locate the online documentation for the **Win32_BIOS** class on the **docs.microsoft.com** website.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

True or False? You use Windows Management Instrumentation (WMI) or Common Information Model (CIM) cmdlets whenever possible because they're easier to use than cmdlets for managing specific object types such as service.

☐ True

☐ False

Question 2

What's the best way to find documentation for WMI and CIM object classes?

☐ Use a search engine to find documentation

☐ Use the `Get-Help` cmdlet

☐ Use the `Get-Member` cmdlet

☐ Use the `Get-Command` cmdlet

Query data by using CIM and WMI

Lesson overview

One of the most common uses for Windows Management Instrumentation (WMI) and Common Information Model (CIM) is querying configuration information from computers. In this lesson, you'll learn more about the structure of the namespaces that contain classes and also how to query instances of a class. You'll learn how to query remote computers by using ad-hoc connections and CIM sessions.

Lesson objectives

After completing this lesson, you'll be able to:

- List the available namespaces.
- List local repository namespaces.
- Retrieve a list of classes from a namespace.
- Retrieve a list of classes from the **root\CIMv2** namespace and sort them.
- Query instances of a specified class.
- Query instances of a specified class by using WMI, CIM, and WMI Query Language (WQL).
- Connect to remote computers by using CIM or WMI.
- Create and manage CIM sessions.
- Query repository classes from remote computers by using CIM sessions objects.

Listing namespaces

Namespaces organize the object classes that you can query with Common Information Model (CIM) and Windows Management Instrumentation (WMI). You can list namespaces to identify potentially useful object classes.

You can use the **Get-WmiObject** cmdlet to list all the namespaces on either the local or a remote computer. To list the namespaces on the local computer, run the following command:

```
Get-WmiObject -Namespace root -List -Recurse | Select -Unique __NAMESPACE
```

Note: If you receive errors when running the previous command, ensure that you're using an elevated Windows PowerShell prompt. Some namespaces have security requirements that allow only administrators to access them.

You can use the **Get-CimInstance** cmdlet to list the namespaces within a specific namespace, but it doesn't provide a -*Recurse* parameter to list all available namespaces in a single command. However, the CIM cmdlets support tab completion for the -*Namespace* parameter, similar to how you can use tab completion when entering a file system path. Using tab completion for the namespace is a browsing method that you can use to explore the available namespaces.

In this module, you'll primarily use the **root\CIMv2** namespace, which includes all the classes related to the Windows operating system and your computer’s hardware. **root\CIMv2** is the default namespace. Therefore, you don't 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'll learn how to list local repository namespaces by using Windows Management Instrumentation (WMI).

Demonstration steps

- Use the **Get-WmiObject** cmdlet to list the WMI namespaces by running the following command:

  Get-WmiObject -Namespace root -List -Recurse | Select -Unique __NAMESPACE

Listing classes

Most of the time, when you use Common Information Model (CIM) and Windows Management Instrumentation (WMI) you're trying to accomplish a specific task. To identify how to accomplish that task, you typically do an internet search to identify if anyone has provided sample code that accomplishes a similar task. Then you can modify that code for your purposes and identify the CIM or WMI classes that they're using. When you don't find useful sample code, you might want to browse the available classes to check if anything is suitable.

To explore the classes available to you by using CIM and WMI, you can list the classes available in a namespace. For example, to list all the classes in the `root\CIMv2` namespace, run either of these commands:

```
Get-WmiObject -Namespace root\CIMv2 -List
```

```
Get-CimClass -Namespace root\CIMv2
```

Windows PowerShell doesn't list classes in any particular order. You can sort the output of your queries alphabetically to find classes more easily. For example, if you want a class that represents a process but don't know the class name, you can quickly refer to the "P" section of a sorted list and start to search for the word *process*. To produce an alphabetical list of classes in the **root\CIMv2** namespace, run either of the following commands:

```
Get-WmiObject -Namespace root\cimv2 -List | Sort Name
```

```
Get-CimClass -Namespace root\CIMv2 | Sort CimClassName
```

Note: In the **root\CIMv2** namespace, you'll notice 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 that's 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 a guessing game. You have to guess what the class might be named and then review the class list to figure out whether you're correct. Then you must query the class to determine whether it contains the information that you need. Because many classes outside the **root\CIMv2** namespace are not well-documented, this is your best approach.

No central directory of repository classes exists. The repository doesn't 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\CIMv2` namespace having *network* in the class name, use the following command:

```
Get-CimClass *network* | Sort CimClassName
```

However, this technique does not provide the ability to search class descriptions because that information isn't stored in the repository. An internet search engine provides more viable alternative to search for possible class names.

Note: You might notice some class names that begin with two underscores (__). These are system classes that WMI and CIM use internally.

Note: There are several free graphical tools that you can use to browse WMI and CIM classes. To find these tools, use a search engine to search for **WMI explorer** or **CIM explorer**.

There's one specific WMI class object that can cause problems for system administrators. This is the **Win32_Product** class. You can use this class to query installed software, but be aware that returning the results takes a long time and has negative performance implications. 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. Microsoft recommends using the **Win32reg_AddRemovePrograms** class as an alternative, but this class is only available on systems with the Microsoft Endpoint Configuration Manager client installed.

Additional reading: You can find the Microsoft Support page for this MSI reconfiguration issue at **Event log message indicates that the Windows Installer reconfigured all installed applications**[3].

Demonstration: Listing and sorting the classes from a CIM namespace

In this demonstration, you'll learn how to list and sort the classes in a namespace.

Demonstration steps

1. Use the **Get-CimClass** cmdlet to list the classes in the **root\SecurityCenter2** namespace.
2. Use the **Get-CimClass** cmdlet to list the classes in the **root\CIMv2** namespace and sort the list by **CimClassName**.
3. Use the **Get-CimClass** cmdlet to list all the classes in the **root\CIMv2** namespace that have **network** in the class name.

Querying instances

Once you have identified the class you want to query, you can use Windows PowerShell to retrieve the specific instances of that class. 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
```

3 https://aka.ms/jlgark

```
Get-CimInstance -ClassName Win32_LogicalDisk
```

Note: The output from these commands is formatted differently, but they contain the same information.

Note: When using **Get-CimInstance**, you can use tab completion for the class name. This isn't possible with **Get-WmiObject**.

Both the -*Class* parameter of **Get-WmiObject** and the -*ClassName* parameter of **Get-CimInstance** are positional. The names of positional parameters don't need to be specified. That means the following commands provide the same results:

```
Get-WmiObject Win32_LogicalDisk
```

```
Get-CimInstance Win32_LogicalDisk
```

Filtering instances

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 don't use Windows PowerShell comparison operators. Instead, they use traditional programming operators, as listed in the following table.

Table 1: Programming operators

Comparison	**WMI and CIM operator**	**Windows PowerShell operator**
Equal to	**=**	**-eq**
Not equal to	**<>**	**-ne**
Greater than	**>**	**-gt**
Less than	**<**	**-lt**
Less than or equal to	**<=**	**-le**
Greater than or equal to	**>=**	**-ge**
Wildcard string match	**LIKE** (with **%** as the wildcard)	**-like** (with ***** as the wildcard)
Require two or more conditions to be true	**AND**	**-and**
Require one of two or more conditions to be true	**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 WMI Query Language (WQL). WQL is a subset of Structured Query Language (SQL) that is specific to querying WMI. Their format is fairly intuitive so it is relatively straightforward to author them. For examle, the following queries are equivalent to the previously described commands that retrieve the specific instance of the **Win32_LogicalDisk** class:

```
Get-WmiObject -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType = 3"


Get-CimInstance -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType =
3"
```

Additional reading: For more information about WQL, refer to **WQL (SQL for WMI)**[4].

Demonstration: Querying class instances

In this demonstration, you'll learn several ways to query class instances from the repository.

Demonstration steps

1. Use the **Get-WmiObject** cmdlets to display all instances of the **Win32_Service** class.
2. Use the **Get-CimInstance** cmdlet to display all instances of the **Win32_Process** class.
3. Use the **Get-CimInstance** cmdlet to filter instances of the **Win32_LogicalDisk** class to display only a **DriveType** of **3**.
4. Use the **Get-CimInstance** cmdlet with the following WQL query to display all instances of the **Win32_NetworkAdapter** class:

   - Query: "SELECT * FROM Win32_NetworkAdapter"

Connecting to remote computers

You can use Windows Management Instrumentation (WMI) and Common Information Model (CIM) cmdlets to query and manage remote computers. When you connect to a remote computer, you can specify alternative credentials for the connection, but alternative credentials are optional. WMI and CIM cmdlets have different capabilities and different syntaxes for remote connections.

Remote connections using WMI cmdlets

For the WMI commands, use the *-ComputerName* parameter to specify a remote computer's name or IP address. You can specify multiple computer names to run the command on multiple computers in a single statement. You can provide the computer names as a comma-separated list, an array containing multiple computer names, or a parenthetical command that produces a collection of computer names as string objects.

Use the *-Credential* parameter to specify an alternative username. If you specify only a username, then you're prompted for the password. If you use the **Get-Credential** cmdlet to store the username and password in a variable, then you can reference that variable to eliminate the password prompt. In the following example, you'll be prompted for the password:

4 https://aka.ms/sql-for-wmi

```
Get-WmiObject -ComputerName LON-DC1 -Credential ADATUM\Administrator -Class
Win32_BIOS
```

When you specify multiple computer names, Windows PowerShell contacts them one at a time in the order that you specify. If connectivity to one computer fails, the command produces an error message and continues to try the remaining computers.

Remote connections using CIM cmdlets

The CIM cmdlets also provide support for ad hoc connections to remote computers by using the *-ComputerName* parameter. However, the CIM cmdlets don't have a *-Credential* parameter to specify alternate credentials. If you want to use alternate credentials, you need to create a CIM session.

You can run the following CIM command to retrieve the same information as the **Get-WmiObject** command in the previous code example:

```
Get-CimInstance -ComputerName LON-DC1 -Classname Win32_BIOS
```

Remember that CIM commands use the WS-MAN protocol for ad hoc connections. This protocol has specific authentication requirements. When establishing a connection between computers in the same domain or in trusting domains, you typically have to provide a computer's name as it displays in Active Directory Domain Services (AD DS). You can't provide an alias name or an IP address because that will result in a failure of Kerberos authentication. You'll learn more about these and other restrictions in Module 8, "Administering remote computers with Windows PowerShell." You'll also learn how to work around these restrictions.

Using CIM sessions

A Common Information Model (CIM) session is a persistent configuration object that's used when creating a connection to a remote computer. The connection uses WS-MAN by default, but you can specify the DCOM protocol. After a session is created, you can use it to process multiple queries for that computer. This simplifies connectivity because all of the configuration options are contained in the session. A CIM session also allows you to specify connectivity options that aren't available for an ad hoc connection.

Creating session objects

When you create a session, you should store it in a variable to reference it later. The basic syntax to create a session and store it in a variable is:

```
$s = New-CimSession -ComputerName LON-DC1
```

You can create multiple sessions at the same time:

```
$sessions = New-CimSession -ComputerName LON-CL1,LON-DC1
```

When you create a session, PowerShell doesn't establish the connection immediately. When a cmdlet uses the CIM session, PowerShell connects to the specified computer, and then, when the cmdlet finishes, PowerShell terminates the connection.

Note: In some cases, it might be beneficial to use PowerShell remoting instead of CIM sessions for remote connectivity. PowerShell remoting opens a connection to the remote computer and keeps it open until explicitly closed. If you're running multiple queries against a computer, this might improve performance.

Using sessions

After you've stored the session in a variable, you reference it with CIM cmdlets by using the -*CimSession* parameter. The following example uses a variable that contains multiple sessions:

```
Get-CimInstance -CimSession $sessions -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'll need to do additional configuration. You'll learn more about that configuration in Module 8, "Administering remote computers with Windows PowerShell."

Note: The help information for some cmdlets such as **Get-SmbShare** states 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.

Configuring session options

A session option object allows you to specify many settings for a session. When you create a new session, you specify the session option object to configure the session. The following example creates a session by using DCOM instead of WS-MAN:

```
$opt = New-CimSessionOption -Protocol Dcom
$DcomSession = New-CimSession -ComputerName LON-DC1 -SessionOption $opt
Get-CimInstance -ClassName Win32_BIOS -CimSession $DcomSession
```

The first line in the preceding code creates a session option object that specifies that the DCOM protocol should be used for connectivity. The second line creates a new session by using that session option object and stores it in a variable. The final line uses the session to query the remote computer defined in the session and return the requested information.

Removing sessions

After you create a session, it remains in memory and available for use until the instance of PowerShell is closed. You can manually remove sessions by using the **Remove-CimSession** cmdlet. The following example removes one or more sessions contained in a variable:

```
$sessions | Remove-CimSession
```

To remove the sessions for a specific remote computer, you can query the sessions for that computer and then remove them, as the following example depicts:

```
Get-CimSession -ComputerName LON-DC1 | Remove-CimSession
```

To remove all sessions, run the following command:

```
Get-CimSession | Remove-CimSession
```

Demonstration: Using CIMSession objects

In this demonstration, you'll learn how to query repository classes from remote computers by using CIMSession objects.

Demonstration steps

1. Use the **New-CimSession** cmdlet to create a Common Information Model (CIM) session to **LON-DC1** and store it in a variable.
2. Use the **Get-CimInstance** cmdlet with the variable storing the CIM session to query the **Win32_OperatingSystem** class.
3. Use the variable storing the CIM session to remove the session.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

When performing a wildcard match in a filter for a Common Information Model (CIM) or Windows Management Instrumentation (WMI) cmdlet, which character is used as the wildcard?

☐ `?`

☐ `=`

☐ `*`

☐ `%`

Question 2

Which cmdlet allows you to create a DCOM connection to query CIM objects from a remote computer?

☐ `New-CimSession`

☐ `Get-CimInstance`

☐ `Get-WmiObject`

☐ `Get-CimClass`

Make changes by using CIM and WMI

Lesson overview

In this lesson, you'll learn to use Common Information Model (CIM) and Windows Management Instrumentation (WMI) to make changes by using methods. The methods available to you vary depending on the type of object. Discovering and understanding these methods is an important step in querying and manipulating the repository information.

Lesson objectives

After completing this lesson, you'll be able to:

- Discover the methods of repository objects.
- Find online documentation for methods.
- Find the methods of the **Win32_Service** class and their documentation.
- Explain how to invoke methods of repository objects.
- Use methods for the **Win32_OperatingSystem** and **Win32_Process** classes.

Discovering methods

A method is an action that you can perform on an object. Repository objects that you query by using Common Information Model (CIM) or Windows Management Instrumentation (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.

When you query instances of a class, you can use the **Get-Member** cmdlet to discover the methods available for that type of object. The following examples depict how to use **Get-Member** to review the properties and methods for instances of **Win32_Service**:

```
Get-WmiObject -Class Win32_Service | Get-Member -MemberType Method

Get-CimInstance -ClassName Win32_Service | Get-Member -MemberType Method
```

You can also use **Get-CimClass** to review the methods available for a specific class:

```
Get-CimClass -Class Win32_Service | Select-Object -ExpandProperty CimClass-
Methods
```

The methods available to managing object instances vary, depending on the type of object. The output of the commands doesn't explain how to use the methods, so unless you already know how to use them, you will need to find the relevant documentation by relying on other sources of information, such as results of an internet search.

Finding documentation for methods

If any method documentation exists, the documentation webpage for the repository class will include it. Remember that repository classes aren't typically well-documented, especially the classes that aren't in the **root\CIMv2** namespace.

An internet search engine provides the fastest way to find the documentation for a class. When you enter the class name into a search engine, one of the first few search results is typically the class documentation page. Generally, the documentation page includes a section named **Methods** that lists the class methods. Select any method name to display the instructions for using that method.

The documentation 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'll learn how to find class methods and how to find their documentation.

Demonstration steps

1. To use Windows Management Instrumentation (WMI) to display the members of the **Win32_Service** class, run the following command:

   Get-WmiObject -ClassName Win32_Service | Get-Member

2. To use Common Information Model (CIM) to display the methods of the **Win32_Service** class, run the following command:

   Get-CimClass -Class Win32_Service | Select-Object -ExpandProperty CimClassMethods

3. To review members returned by the **Get-CimClass** cmdlet, run the following command:

   Get-CimClass -Class Win32_Service | Get-Member

4. On the host machine, open a browser, refer to **docs.microsoft.com**, and then find online documentation for the **Change** method of **Win32_Service**.

Invoking methods

To use a method on an object, you invoke it. You can accomplish this by using direct invocation, which includes a reference to the object that supports the method, followed by the method name. Alternatively, you can use the **Invoke-WmiMethod** or **Invoke-CimMethod** cmdlets. If you use the cmdlets, the cmdlet you use needs to match the type of object you're working with.

Direct invocation

Direct invocation is typically used when you've loaded Windows Management Instrumentation (WMI) objects into a variable. Then, you can invoke methods available for that object type by specifying the variable name, a dot (**.**), and then the method. This is similar to how you display property values for an object contained in a variable. The following example queries the spooler service as a WMI object and then invokes the **StopService** method.

```
$WmiSpoolerService = Get-WmiObject -Class Win32_Service -Filter
"Name='Spooler'"
$WmiSpoolerService.StopService()
```

The **StopService** method in the previous example doesn't require any parameters to be passed to it, so there's no value within the parentheses. If you invoke a method that requires parameters, their values are placed within the parentheses. The following example sets the value of the start mode parameter to **Manual**:

```
$WmiSpoolerService.ChangeStartMode("Manual")
```

To identify the parameters required for a method, you should review the documentation for its class. Be aware that parameters for WMI method parameters need to be passed in a specific order. To identify the order, you can use the **GetMethodParameters()** method. The following example queries the parameters for the **Change** method:

```
$WmiSpoolerService.GetMethodParameters("Change")
```

If the method requires multiple parameters and you don't want to change some of them, you can pass a `$null` value for the parameters you don't want to change. Additionally, you don't need to specify parameters that are positioned after the one you want to change. In the following example, the Change method has 11 parameters, but only the second parameter (display name) is being configured.

```
$WimSpoolerService.Change($null,"Printer Service")
```

Note: Because `$null` is treated as a parameter value to skip when calling a method, you can't set a value to `$null` by invoking a method on a WMI object.

Note: Objects retrieved by **Get-CimInstance** are static and don't have methods. Therefore, you can't use them by relying on direct invocation.

Using the Invoke-WmiMethod cmdlet

The **Invoke-WmiMethod** cmdlet is another way to invoke a method for a WMI object with different syntax. The -*Name* parameter is used to specify the method to invoke and the -*Argument* parameter is used to specify the parameter values that are passed to the method. If required, multiple parameters are passed as a comma-separated list or array. The parameter values need to be in a specific order just as they were for direct invocation.

You can use the **Invoke-WmiMethod** cmdlet by itself, or you can use the pipeline to send it a WMI object. Here are two examples that work the same way:

```
Get-WmiObject -Class Win32_OperatingSystem | Invoke-WmiMethod -Name Win-
32Shutdown -Argument 0


Invoke-WmiMethod -Class Win32_OperatingSystem -Name Win32Shutdown -Argument
0
```

The **Get-WmiObject** and **Invoke-Method** cmdlets both have the -*ComputerName* parameter that lets you run the cmdlet on a remote computer.

Using the Invoke-CimMethod cmdlet

The **Invoke-CimMethod** cmdlets provide similar functionality to the **Invoke-WmiMethod** cmdlet, but 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 parameter name, and the value for each pair is the corresponding parameter value. In the following example, **Path** is the name of the parameter and **Notepad.exe** is the value:

```
Invoke-CimMethod -ComputerName LON-DC1 -ClassName Win32_Process -MethodName
Create -Arguments @{CommandLine='Notepad.exe'}
```

To include multiple parameters in the dictionary, use a semicolon to separate each key-value pair. Because the parameters are named, the key-value pairs can be in any order.

To invoke a method on a specific instance of an object, you retrieve the object first by using the **Get-CimInstance** cmdlet. You can pipe the object directly to the **Invoke-CimMethod** cmdlet or store it in a variable first. If the object provides all of the necessary information, then you don't need to specify any arguments. The following example retrieves any running instances of Notepad.exe and terminates them:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name='notepad.exe'" |
Invoke-CimMethod -MethodName Terminate
```

If you use the *-ComputerName* or *-CIMSession* parameters with the **Get-CimInstance** cmdlet and pipe the resulting object to the **Invoke-CimMethod** cmdlet, the method is invoked 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
```

Demonstration: Invoking methods of repository objects

In this demonstration, you'll learn how to invoke methods of repository objects.

Demonstration steps

1. From **LON-CL1**, use the **Invoke-CimMethod** cmdlet to invoke the **Reboot** method of the **Win32_OperatingSystem** class on **LON-DC1**.
2. Use the **Invoke-CimMethod** cmdlet to start **mspaint.exe** by invoking the **Create** method of the **Win32_Process** class.
3. To use the **Terminate** method of the **Win32_Process** class to close Microsoft Paint, run the following command:

   Get-CimInstance -Class Win32_Process -Filter "Name='mspaint.exe'" | Invoke-CimMethod -Name Terminate

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which property returned by the `Get-CimClass` *cmdlet lists the methods supported by that class?*

☐ CimClassProperties

☐ CimClassQualifiers

☐ CimClassMethods

☐ CimSystemProperties

Question 2

True or False? You can use direct invocation for methods on Windows Management Instrumentation (WMI) and CIM objects.

☐ True

☐ False

Module 05 lab and review

Lab: Querying information by using WMI and CIM

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'll be able to:

- Query information by using Windows Management Instrumentation (WMI) commands.
- Query information by using Common Information Model (CIM) commands.
- Invoke methods by using WMI and CIM commands.

Estimated time: 45 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

Which port number is used for connectivity to a WS-MAN listener?

☐ 80

☐ 443

☐ 5985

☐ 8080

Question 2

What type of object is a specific service that you've retrieved by using CIM?

☐ Instance

☐ Class

☐ Namespace

☐ Listener

Question 3

Which cmdlets can you use to query classes in the `root\CIMv2` *namespace? Choose two.*

- ☐ `Get-WmiObject`
- ☐ `Get-CimInstance`
- ☐ `Get-CimClass`
- ☐ `Invoke-WmiObject`
- ☐ `Invoke-CimObject`

Question 4

Which cmdlets can you use to query a list of disk drives in the `root\CIMv2` *namespace? Choose two.*

- ☐ `Get-WmiObject`
- ☐ `Get-CimInstance`
- ☐ `Get-CimClass`
- ☐ `Invoke-WmiObject`
- ☐ `Invoke-CimObject`

Question 5

True or False? When you pass method parameters by using CIM, the parameters can be in any order.

- ☐ True
- ☐ False

Answers

Question 1

True or False? You use Windows Management Instrumentation (WMI) or Common Information Model (CIM) cmdlets whenever possible because they're easier to use than cmdlets for managing specific object types such as service.

☐ True

■ False

Explanation
False is the correct answer. For managing a specific type of object, cmdlets are typically easier to use than the WMI or CIM cmdlets.

Question 2

What's the best way to find documentation for WMI and CIM object classes?

■ Use a search engine to find documentation

☐ Use the `Get-Help` cmdlet

☐ Use the `Get-Member` cmdlet

☐ Use the `Get-Command` cmdlet

Explanation
Use a search engine to find documentation is the correct answer. Windows PowerShell doesn't include any help that's specific to WMI and CIM classes. Microsoft provides documentation for many WMI and CIM classes. Finding help for a specific class is often fastest by using a search engine.

Question 1

When performing a wildcard match in a filter for a Common Information Model (CIM) or Windows Management Instrumentation (WMI) cmdlet, which character is used as the wildcard?

☐ `?`

☐ `=`

☐ `*`

■ `%`

Explanation
% is the correct answer. When you use LIKE in a filter for CIM or WMI, the `%` character is used as the wildcard character. Typically, Windows PowerShell uses the `*` character.

Question 2

Which cmdlet allows you to create a DCOM connection to query CIM objects from a remote computer?

■ `New-CimSession`

☐ `Get-CimInstance`

☐ `Get-WmiObject`

☐ `Get-CimClass`

Explanation
New-CimSession is the correct answer. By default, the CIM cmdlets use WS-MAN for connectivity to remote computers. You can force a remote connection to use DCOM by creating a CIM session object with the DCOM setting and then using the CIM session to perform the query.

Question 1

Which property returned by the `Get-CimClass` cmdlet lists the methods supported by that class?

☐ CimClassProperties

☐ CimClassQualifiers

■ CimClassMethods

☐ CimSystemProperties

Explanation
CimClassMethods is the correct answer. The CimClassMethods property contains a list of methods that can be invoked for the object class.

Question 2

True or False? You can use direct invocation for methods on Windows Management Instrumentation (WMI) and CIM objects.

☐ True

■ False

Explanation
False is the correct answer. The CIM cmdlets use WS-MAN and return static objects. You can't use direct invocation on a static object.

Question 1

Which port number is used for connectivity to a WS-MAN listener?

☐ 80

☐ 443

■ 5985

☐ 8080

Explanation
5985 is the correct answer. A WS-MAN listener is automatically created on port 5985. If a certificate is installed on the computer, a listener using TLS can be created on port 5986.

Question 2

What type of object is a specific service that you've retrieved by using CIM?

■ Instance

☐ Class

☐ Namespace

☐ Listener

Explanation
The correct answer is instance. A class is a type of object and an instance is a specific occurrence of a class.

Question 3

Which cmdlets can you use to query classes in the `root\CIMv2` namespace? Choose two.

■ `Get-WmiObject`

☐ `Get-CimInstance`

■ `Get-CimClass`

☐ `Invoke-WmiObject`

☐ `Invoke-CimObject`

Explanation
The correct answers are `Get-WmiObject` *and* `Get-CimClass`.

Question 4

Which cmdlets can you use to query a list of disk drives in the `root\CIMv2` namespace? Choose two.

■ `Get-WmiObject`

■ `Get-CimInstance`

☐ `Get-CimClass`

☐ `Invoke-WmiObject`

☐ `Invoke-CimObject`

Explanation
The correct answers are `Get-WmiObject` *and* `Get-CimInstance`.

Question 5

True or False? When you pass method parameters by using CIM, the parameters can be in any order.

■ True

☐ False

Explanation
True is the correct answer. When you use the `Invoke-CimMethod` *cmdlet, the parameters are named and can be passed in any order.*

Module 6 Working with variables, arrays, and hash tables

Use variables

Lesson overview

In this lesson, you'll learn about variables and some rules for how to use them. These rules help ensure that the data you store in variables is in the correct format and easily accessible. When working with variables, it's important to name them appropriately and to assign them the correct data type.

Lesson objectives

After completing this lesson, you'll 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.
- Modifying the objects.
- Displaying the data.

The pipeline's primary limitation is that the process flows only in one direction and it's difficult to perform complex operations. You can use variables to solve this problem. Variables store values and objects in memory so you can perform complex and repetitive operations on them.

Some of the tasks you can do with a variable are:

- Storing the name of a log file that you write data to multiple times.
- Deriving and storing an email address based on the name of a user account.
- Calculating and storing the date representing the beginning of the most recent 30-day period, to identify whether computer accounts have authenticated during that time.

In addition to simple data types such as numbers or strings, variables can contain 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 that user account's properties are stored in the variable as well, and you can review them. To review the variables contained in memory by reviewing the contents of the PSDrive named **Variable**, use the following command:

```
Get-ChildItem Variable:
```

You can also review the variables in memory by using the **Get-Variable** cmdlet:

```
Get-Variable
```

Note: Windows PowerShell includes multiple cmdlets for creating, manipulating, and reviewing variables. However, these are seldom used because you can create and manipulate variables directly, without resorting to using cmdlets. Therefore, this course only briefly mentions cmdlets for manipulating variables.

Variable naming

You should create variable names that describe the data stored in them. For example, a variable that stores a user account could be `$user`, and a variable that stores the name of a log file could be `$logFileName`.

In most cases, you'll notice variables are used with a dollar sign (`$`) symbol. The `$` symbol is not part of the variable name but it distinguishes variables from other syntax elements of Windows PowerShell. For example, `$user` designates a variable named **user**, and the `$` symbol helps Windows PowerShell to identify that it's 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. For example, to include a space in a variable name, you need to enclose the name in braces (`{ }`). An example would be `${log File}`, where there's a space between the words log and file.

Variable names aren't case sensitive. The variables `$USER` and `$user` are interchangeable. For improved legibility, the common convention is to use lowercase characters and capitalize the first letter of each word in a variable name. Capitalizing 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 more legible without using special characters such as spaces, hyphens, or underscores.

Assigning a value to a variable

When working with variable values, you use the standard mathematical operators that you're already familiar with, such as equal (=), plus (+), and minus (−).

To assign a value to a variable, you use the equal (=) operator. For example:

```
$num1 = 10
$logFile = "C:\Logs\log.txt"
```

You can also assign a value to a variable by using a command that's evaluated. The result of the command is placed in the variable. For example:

```
$user = Get-ADUser Administrator
$service = Get-Service W32Time
```

Note: If a command returns multiple results, then the variable becomes an array containing multiple values. You'll learn about arrays later in this module.

You can display the value of a variable by entering the variable name and then pressing the Enter key. 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** and you place the variable name in double quotes ("), that variable is evaluated and its value is displayed. If you use single quotes ('), that variable is not evaluated, resulting in its name being displayed instead.

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 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, as the following example depicts:

```
$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 don't include the `$` symbol when referring to the name, as the following example depicts:

```
Set-Variable -Name num1 -Value 5
```

Additional reading: For more information about assignment operators, refer to **about_Assignment_Operators**[1].

1 https://aka.ms/lewact

Variable types

All variables are assigned a type. The variable type determines the data that can be stored in it. In most cases, Windows PowerShell automatically determines the type of a variable during assignment of its 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 set the variable type explicitly.

The following table lists the common variable types used in Windows PowerShell.

Table 1: Common variable types used in Windows PowerShell

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, 2022 10:00 AM**.
Bool	A Boolean variable can store only the values `$true` or `$false`.

If you don't assign a variable type, Windows PowerShell assigns a type automatically based on the value you assign to the variable. When the value is contained in quotes, it's generally interpreted as a string. For example, Windows PowerShell would interpret **5** as an integer but **"5"** would be interpreted 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 following examples depict 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, 2022 10:00AM"
```

You can review 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'll learn how to assign values and types to variables.

Demonstration steps

1. On **LON-CL1**, open Windows PowerShell.

2. To set the value of `$num1` to **5**, at the Windows PowerShell prompt, run the following command:

   ```
   $num1 = 5
   ```

3. To display the value of `$num1`, run the following command:

   ```
   $num1
   ```

4. To set the value of `$logFile` to **C:\Logs\Log.txt**, run the following command:

   ```
   $logFile = "C:\Logs\Log.txt"
   ```

5. To review the value of `$logFile`, run the following command:

   ```
   $logFile
   ```

6. To set the value of `$service` to the **W32Time** service, run the following command:

   ```
   $service = Get-Service W32Time
   ```

7. To display the value of `$service`, run the following command:

   ```
   $service
   ```

8. To display `$logFile` as part of a text message on screen, run the following command:

   ```
   Write-Host "The log file location is $logFile"
   ```

9. To review all of the properties of the service object stored in `$service`, run the following command:

   ```
   $service | Format-List *
   ```

10. To review the **Status** property of `$service`, run the following command:

    ```
    $service.status
    ```

11. To review the **Name** and **Status** properties of `$service`, run the following command:

    ```
    $service | Format-Table Name,Status
    ```

12. To review the variables in memory, run the following command:

    ```
    Get-Variable
    ```

13. To review the variables in memory, run the following command:

    ```
    Get-ChildItem Variable:
    ```

14. To review the variable type of `$num1`, run the following command:

    ```
    $num1.GetType()
    ```

15. To review the variable type of `$logFile`, run the following command:

```
$logFile.GetType()
```

16. To review the variable type of `$service`, run the following command:

```
$service.GetType()
```

17. To review the properties and methods of `$service`, run the following command:

```
$service | Get-Member
```

18. To set the value of `$num2` as a string of **5**, run the following command:

```
$num2 = "5"
```

19. To set the value of `$num3` as a 32-bit integer of **5**, run the following command:

```
[Int]$num3 = "5"
```

20. To verify the variable type of `$num2`, run the following command:

```
$num2.GetType()
```

21. To verify the variable type of `$num3`, run the following command:

```
$num3.GetType()
```

22. To set `$date1` as a string, run the following command:

```
$date1 = "March 5, 2019 11:45 PM"
```

23. To set `$date2` as a DateTime type, run the following command:

```
[DateTime]$Date2 = "March 5, 2019 11:45 PM"
```

24. To verify the variable type of `$date1`, run the following command:

```
$date1.GetType()
```

25. To verify the variable type of `$date2`, run the following command:

```
$date2.GetType()
```

26. To attempt to convert a string to a 32-bit integer, run the following command:

```
[Int]$num4 = "Text that can't convert"
```

27. To review how variable types can convert during operations, run the following command:

```
$num2 + $num3
```

28. To review how variable types can convert during operations, run the following command:

```
$num3 + $num2
```

29. To review how variable types can fail to convert during operations, run the following command:

```
$num3 + $logFile
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which character is used to identify variables?

- ☐ `#`
- ☐ `%`
- ☐ `!`
- ☐ `$`

Question 2

Which command would you use to assign a numerical value of 20 to a variable?

- ☐ $num = 20
- ☐ $num = "20"
- ☐ $num -eq 20
- ☐ $num -eq "20"

Manipulate variables

Lesson overview

In this lesson, you'll learn how to identify the properties and methods for use in your scripts. You'll also 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.

Lesson objectives

After completing this lesson, you'll be able to:

- Identify methods and properties.
- Work with strings.
- Manipulate strings.
- Work with dates.
- 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's 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 that are 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
```

To browse through the properties and methods for a variable, you leverage the tab completion feature by entering the name of the variable appended with a dot. When you select the **Tab** key, the properties and methods available for the variable display.

When you review the properties and methods for a variable, some of them will be easily understandable. If you don't understand how to use a property or method for a variable, 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 Struct.

Additional reading: For more information on .NET Framework variable types, refer to the **Structs** section at **System Namespace**[2].

The documentation for the .NET Framework is oriented towards developers and sometimes 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.

2 https://aka.ms/system-namespace

Working with strings

String variables are commonly used in scripts. You can use strings to store user input and other text data. There are many methods that you can use to manipulate strings. Many of these methods are seldom used, but it's good to be aware of them in case you ever need them.

The string variable has only one property, **Length**. When you review 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.

Table 1: 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 isn't specified, then the string is truncated at the 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.

Demonstration: Manipulating strings

In this demonstration, you'll learn how to manipulate string variables.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt with administrative permissions.
2. To set `$logFile` with a value, at the Windows PowerShell prompt, run the following command:

   ```
   $logFile = "C:\Logs\log.txt"
   ```

3. To identify whether `$logFile` contains the text **C:**, run the following command:

   ```
   $logFile.Contains("C:")
   ```

4. To identify whether `$logFile` contains the text **D:**, run the following command:

   ```
   $logFile.Contains("D:")
   ```

5. To insert the text **\MyScript** at character **7**, run the following command:

   ```
   $logFile.Insert(7,"\MyScript")
   ```

6. To verify that the value stored in `$logFile` hasn't changed, run the following command:

   ```
   $logFile
   ```

7. To update the value of `$logFile`, run the following command:

   ```
   $logFile=$logFile.Insert(7,"\MyScript")
   ```

8. To verify that the value of `$logFile` has changed, run the following command:

   ```
   $logFile
   ```

9. To replace **.txt** with **.htm**, run the following command:

   ```
   $logFile.Replace(".txt",".htm")
   ```

10. To split the value of `$logFile` at the `\` character, run the following command:

    ```
    $logFile.Split("\")
    ```

11. To review only the last item from the split, run the following command:

    ```
    $logFile.Split("\") | Select -Last 1
    ```

12. To convert the value to uppercase letters, run the following command:

    ```
    $logFile.ToUpper()
    ```

13. To convert the value to lowercase letters, run the following command:

    ```
    $logFile.ToLower()
    ```

Working with dates

Many scripts that you create will need to reference the current date or a previous point in time. For example, to ensure uniqueness, you might want to create a log file name based on the current date. Additionally, you might be searching for users in AD DS that haven't 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 DateTime variable properties to access specific parts of the date or time. The following table lists some of the properties available for a DateTime variable.

Table 1: 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 following table lists some of the DateTime variable methods.

Table 2: DateTime variable methods

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.
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 as its parameter. An example is `$date.AddDays(-60)`.

Demonstration: Manipulating dates

In this demonstration, you'll learn how to manipulate DateTime variables.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt with administrative permissions.
2. To put the current date and time in the variable `$date`, at the Windows PowerShell prompt, run the following command:

```
$date = Get-Date
```

3. To display the value of `$date`, at the Windows PowerShell prompt, run the following command:

```
$date
```

4. To display the **Hour** property of `$date`, run the following command:

   ```
   $date.Hour
   ```

5. To display the **Minute** property of `$date`, run the following command:

   ```
   $date.Minute
   ```

6. To display the **Day** property of `$date`, run the following command:

   ```
   $date.Day
   ```

7. To display the **DayOfWeek** property of `$date`, run the following command:

   ```
   $date.DayOfWeek
   ```

8. To display the **Month** property of `$date`, run the following command:

   ```
   $date.Month
   ```

9. To display the **Year** property of `$date`, run the following command:

   ```
   $date.Year
   ```

10. To add **100** days to `$date`, run the following command:

    ```
    $date.AddDays(100)
    ```

11. To subtract **60** days from `$date`, run the following command:

    ```
    $date.AddDays(-60)
    ```

12. To display `$date` as a long date string, run the following command:

    ```
    $date.ToLongDateString()
    ```

13. To display `$date` as a short date string, run the following command:

    ```
    $date.ToShortDateString()
    ```

14. To display `$date` as a long time string, run the following command:

    ```
    $date.ToLongTimeString()
    ```

15. To display `$date` as a short time string, run the following command:

    ```
    $date.ToShortTimeString()
    ```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

The variable `$ip` *is a string with a value of "10.100.4.2". Which command would you use to divide* `$ip` *into four separate numbers?*

☐ $ip.Divide(4)

☐ $ip.Split(4)

☐ $ip.Split(".")

☐ $ip | Break 4

Question 2

Which of the following options are methods that you can use with DateTime variables? Choose three.

☐ AddDays(double value)

☐ AddMinutes(double value)

☐ ToLower()

☐ ToShortDateString()

☐ DayOfWeek

Manipulate arrays and hash tables

Lesson overview

You can use arrays and hash tables to store data that's more complex than simple variables, and to complete more complex tasks with your scripts. In this lesson, you'll learn how to use arrays and hash tables.

Lesson objectives

After completing this lesson, you'll be able to:

- Explain the purpose of an array.
- Work with arrays and their contents.
- Work with array lists and their contents.
- Manipulate arrays and array lists.
- Explain the purpose of a hash table.
- Work with hash tables and their contents.
- Manipulate hash tables.

What is an array?

An *array* is a data structure that's designed to store a collection of items of the same type. You can think of an array as a variable containing 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 domain user accounts. Or you might need to check the status of a group of services and restart all of them 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's 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 **Base-Type** listed will be **System.Array**.

You can create an empty array before you're 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

Within your scripts, you'll need to refer to the data that you place in arrays. You can either access all items in the array simultaneously or access them individually. To display all items in an array, you enter the variable name and then press the Enter key, 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 0. So, the first item placed in the array is item 0, the second item in the array is item 1, and so on. To display a specific item, place the index number in brackets after the variable name. The following example displays the first item in an array that's stored in the variable $users:

```
$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
```

Alternatively, when adding items to an array, you can use the following shorthand notation:

```
$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. This is also a helpful way of determining which data types are in the array.

To review the properties and methods available for an array rather than the items within the array, use the following syntax:

```
Get-Member -InputObject $files
```

Working with array lists

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, recreating an array each time has a negative performance impact. The other concern when using fixed-size arrays is removing items. There's no simple method to remove an item from a fixed-size array.

To address the shortcomings of arrays, you can use an array list. An array list functions similar to an array, except that it doesn't have a fixed size. This means that you can use methods to add and remove items.

To create an array list when assigning values, use the following syntax:

```
[System.Collections.ArrayList]$computers = "LON-DC1","LON-SVR1","LON-CL1"
```

To create an array list that's empty and ready to add items, use the following syntax:

```
$computers=New-Object System.Collections.ArrayList
```

When you use an array list, 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 array list, if there are multiple matching items then only the first instance is removed.

If you want to remove an item from an array list based on the index number, you use the **RemoveAt()** method. For example:

```
$computers.RemoveAt(1)
```

Demonstration: Manipulating arrays and array lists

In this demonstration, you'll learn how to manipulate arrays and array lists.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt with administrative permissions.
2. To set `$computers` to be an array of strings, at the Windows PowerShell prompt, run the following command:

   $computers = "LON-DC1","LON-SRV1","LON-CL1"

3. To set `$users` to be an array of user objects, run the following command:

   $users = Get-ADUser -Filter *

4. To review the contents of the `$computers` array, run the following command:

   $computers

5. To review the contents of the `$users` array, run the following command:

   $users

6. To review the number of items in `$users`, run the following command:

   $users.count

7. To review the user object at index **125** of `$users`, run the following command:

   ```
   $users[125]
   ```

8. To review the properties and methods available for the items in `$computers`, run the following command:

   ```
   $computers | Get-Member
   ```

9. To review the properties and methods available for the items in `$users`, run the following command:

   ```
   $users | Get-Member
   ```

10. To review the **UserPrincipalName** property for a user object in the array, run the following command:

    ```
    $users[125].UserPrincipalName
    ```

11. To add an item to `$computers`, run the following command:

    ```
    $computers += "LON-SRV2"
    ```

12. To verify that the item was added, run the following command:

    ```
    $computers
    ```

13. To create an arraylist containing user objects, run the following command:

    ```
    [System.Collections.ArrayList]$usersList = Get-ADUser -Filter *
    ```

14. To identify whether `$usersList` has a fixed size, run the following command:

    ```
    $usersList.IsFixedSize
    ```

15. To review the number of items in `$arrayList`, run the following command:

    ```
    $usersList.count
    ```

16. To review a single item in `$arrayList`, run the following command:

    ```
    $usersList[125]
    ```

17. To remove an item in `$arrayList`, run the following command:

    ```
    $usersList.RemoveAt(125)
    ```

18. To verify that the item count is reduced by one, run the following command:

    ```
    $usersList.count
    ```

19. To verify that the item at index **125** has changed, run the following command:

    ```
    $usersList[125]
    ```

What is a hash table?

A hash table represents a similar concept to an array since it stores multiple items. However, unlike an array which uses an index number to identify each item, a hash table uses for this purpose a unique key. The *key* is a string that's a unique identifier for the item. Each key in a hash table is associated with a value.

The following table depicts how an array can store a list of IP addresses.

Table 1: How an array stores a list of IP addresses

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 the following table depicts.

Table 2: Using a hash table to store IP addresses and computer names

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 either of the following options:

```
$servers.'LON-DC1'
$servers['LON-DC1']
```

Note: You only need to use single quote marks to enclose keys that contain special characters. In the previous example, the hyphen in the computer names is a special character, which requires the key name to be enclosed in single quote 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:

```
$servers = @{"LON-DC1" = "172.16.0.10"; "LON-SRV1" = "172.16.0.11"}
```

Notice the following syntax in the previous example:

- It begins with the at (@) 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 in the previous example because they're all on the same line. If you place each item on a separate line, it's not necessary to use semicolons as separators.

Adding or removing items from a hash table is similar to an array list. 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 review 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'll learn how to manipulate hash tables.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt with administrative permissions.
2. To create a hash table with the names of users and a department for each, at the Windows PowerShell prompt, run the following command:

   $users = @{"Lara"="IT";"Peter"="Managers";"Sang"="Sales"}

3. To review the contents of the hash table, run the following command:

   $users

4. To review the department for a single user, run the following command:

   $users.Lara

5. To update the department for a user, run the following command:

   $users.Sang = "Marketing"

6. To verify that the department was updated, run the following command:

   $users

7. To add a new user, run the following command:

   $users.Add("Tia","Research")

8. To remove a user, run the following command:

```
$users.Remove("Sang")
```

9. To verify the added and removed users, run the following command:

```
$users
```

10. To create a new hash table for a calculated property, run the following command:

```
$prop = @{n="Size(KB)";e={$_.Length/1KB}}
```

11. To review the hash table, run the following command:

```
$prop
```

12. To review the name and size of the files in **C:\Windows**, run the following command:

```
Get-ChildItem C:\Windows -File | Format-Table Name,Length
```

13. To review the size of files by using the calculated properly, run the following command:

```
Get-ChildItem C:\Windows -File | Format-Table Name,$prop
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

How do you refer to the first item in an array?

- ☐ $users[0]
- ☐ $users[1]
- ☐ $users["1"]
- ☐ You cannot refer to items in an array by location.

Question 2

Which command would you use to update a value associated with a key named `DC1` *in a hash table?*

- ☐ $computers(key=dc1;value="Domain Controller")
- ☐ $computers.keys("dc1") = "Domain Controller"
- ☐ $computers[5] = "Domain Controller"
- ☐ $computers.dc1 = "Domain Controller"

Module 06 lab and review

Lab: Using variables, arrays, and hash tables in PowerShell

Scenario

You're preparing to create scripts to automate server administration in your organization. Before you begin, you want to practice working with variables, arrays, and hash tables.

Objectives

After completing this lab, you'll be able to:

- Work with variable types.
- Use arrays.
- Use hash tables.

Estimated time: 45 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

Which command can you use to add an item to an array?

☐ $computers.add("DC1")

☐ $computers.new("DC1")

☐ $computers += "DC1"

☐ $computers | "DC1"

Question 2

Which of the following options are examples of good variable names? (Select two.)

☐ $COMPUTERNAME

☐ $i

☐ $computerName

☐ $num

☐ $ipAddress

Question 3

Which cmdlet should you use to review the properties and methods available for the variable's contents?

- ☐ Select-Object
- ☐ Get-Member
- ☐ GetType()
- ☐ Get-Variable

Question 4

True or False? An array list provides better performance than an array when working with many items.

- ☐ True
- ☐ False

Question 5

Which commands display a variable's contents? (Select two.)

- ☐ $computer
- ☐ Write-Host "This is the computer name: $computer"
- ☐ Write-Host 'This is the computer name: $computer'
- ☐ $computer.chars

Answers

Question 1

Which character is used to identify variables?

☐ `#`

☐ `%`

☐ `!`

■ `$`

Explanation
`$` is the correct answer. In Windows PowerShell, you identify variables by including a `$` at the start of the name.

Question 2

Which command would you use to assign a numerical value of 20 to a variable?

■ $num = 20

☐ $num = "20"

☐ $num -eq 20

☐ $num -eq "20"

Explanation
`$num = 20` is the correct answer. To assign a value to a variable, you must use the equal (=) operator. When you use the value `20`, it's automatically interpreted as a number. If you use `"20"` as the value, it's interpreted as a string.

Question 1

The variable `$ip` is a string with a value of "10.100.4.2". Which command would you use to divide `$ip` into four separate numbers?

☐ $ip.Divide(4)

☐ $ip.Split(4)

■ $ip.Split(".")

☐ $ip | Break 4

Explanation
`$ip.Split(".")` is the correct answer. You use the Split method to divide a string into parts. To use the split method, you need to provide the character that acts as the separator. In this case, dot "." is the separator.

Question 2

Which of the following options are methods that you can use with DateTime variables? Choose three.

- ■ AddDays(double value)
- ■ AddMinutes(double value)
- ☐ ToLower()
- ■ ToShortDateString()
- ☐ DayOfWeek

Explanation
The correct answers are `AddDays(double value)`,`AddMinutes(double value)`, *and* `ToShortDateString()`.

Question 1

How do you refer to the first item in an array?

- ■ $users[0]
- ☐ $users[1]
- ☐ $users["1"]
- ☐ You cannot refer to items in an array by location.

Explanation
`$users[0]` *is the correct answer. In an array, all items have an index number, and the first index number is 0.*

Question 2

Which command would you use to update a value associated with a key named `DC1` in a hash table?

- ☐ $computers(key=dc1;value="Domain Controller")
- ☐ $computers.keys("dc1") = "Domain Controller"
- ☐ $computers[5] = "Domain Controller"
- ■ $computers.dc1 = "Domain Controller"

Explanation
`$computers.dc1 = "Domain Controller"` *is the correct answer. When assigning a value to a key in a hash table, you use the format $hash.key = value.*

Question 1

Which command can you use to add an item to an array?

- ☐ $computers.add("DC1")
- ☐ $computers.new("DC1")
- ■ $computers += "DC1"
- ☐ $computers | "DC1"

Explanation
`$computers += "DC1"` *is the correct answer. The += operator is a shortened version of* `$computers = $computers + "DC1"`.

Question 2

Which of the following options are examples of good variable names? (Select two.)

☐ $COMPUTERNAME

☐ $i

■ $computerName

☐ $num

■ $ipAddress

Explanation
The correct answers are `$computerName` *and* `$ipAddress`. *Both of these variable names are descriptive and use capitalized letters to differentiate between words in the name.*

Question 3

Which cmdlet should you use to review the properties and methods available for the variable's contents?

☐ Select-Object

■ Get-Member

☐ GetType()

☐ Get-Variable

Explanation
`Get-Member` *is the correct answer. When you pipe a variable to* `Get-Member`, *the properties and methods for the variable contents are displayed.*

Question 4

True or False? An array list provides better performance than an array when working with many items.

■ True

☐ False

Explanation
The correct answer is True. An array list is not a fixed size. This means you can use the `Add()` *and* `Re-move()` *methods, which are more efficient than working with fixed size arrays.*

Question 5

Which commands display a variable's contents? (Select two.)

■ $computer

■ Write-Host "This is the computer name: $computer"

☐ Write-Host 'This is the computer name: $computer'

☐ $computer.chars

Explanation
The correct answers are `$computer` *and* `Write-Host "This is the computer name: $com-puter"`. *When a variable name is not enclosed in quotes or is in double-quotes, the variable is evaluated, and the contents are used.*

Module 7 Windows PowerShell scripting

Introduction to scripting with Windows PowerShell

Lesson overview

Most administrators start working with scripts by either downloading or modifying scripts that others create. After you've 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'll 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.
- Set the script execution policy.
- Explain how to use AppLocker to help secure Windows PowerShell scripts.
- Explain how to digitally sign scripts.
- Digitally sign a Windows PowerShell script.

Overview of Windows PowerShell scripts

You might first start using Windows PowerShell to accomplish tasks that you can't do with graphical tools. For example, when managing Microsoft 365 or Microsoft Exchange Server, there are many settings that you can only configure by using Windows PowerShell cmdlets. As you become more familiar with Windows PowerShell, you'll notice 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. If a script has been tested, you can run it multiple times without errors. However, when you manually enter a command multiple times, there's 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.

Note: Module 11, "Using background jobs and scheduled jobs" covers the configuration of Windows PowerShell scripts to make them run as scheduled tasks or scheduled jobs.

You can also use scripts to accomplish more complex tasks than are practical by using a single command. While it's technically possible to make a single Windows PowerShell command that's long and complex, it's 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**, which 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's been saved with the .ps1 file extension. While the Windows PowerShell Integrated Scripting Environment (ISE) and Microsoft Visual Studio Code have better features, you can edit Windows PowerShell scripts by using a simple text editor such as Notepad.

Modifying scripts

Most administrators don't 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 haven't signed in for an extended period of 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 didn't 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: For more information, refer to the **PowerShell Gallery**[1].

1 https://aka.ms/ue14hl

The internet

There are 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 useful resources in the blogs and articles, but you should be very careful in your testing.

Creating scripts

If you can't find a script that meets your needs, you can create your own script. You can still use resources such as the PowerShell Gallery and the internet to find code snippets that are useful in building that script, so you don't 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's likely that you'll 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're 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're 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.

What is the PowerShellGet module?

The **PowerShellGet** module includes cmdlets for accessing and publishing items in the PowerShell Gallery. This module was introduced in Windows Management Framework 5.0, which is included in Windows 10 and Windows Server 2016. You can upgrade older Windows operating systems to include Windows Management Framework 5.0 or newer 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 you can use on systems with Windows PowerShell 4.0.

When you use the cmdlets in the **PowerShellGet** module for the first time, you're 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.

The following table lists the two cmdlets used most often to find content in the PowerShell Gallery.

Table 1: Cmdlets used to find content in the PowerShell Gallery

Cmdlet	Description
Find-Module	Use this cmdlet 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.

Cmdlet	Description
Find-Script	Use this cmdlet 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: You can also search for modules and scripts in the **PowerShell Gallery**[2].

TLS 1.2

The PowerShell Gallery requires the use of Transport Layer Security (TLS) 1.2 to help secure communication. Windows 10 and Windows Server 2016 don't support using TLS 1.2 in Windows PowerShell by default. So, you need to enable TLS 1.2 to download PowerShell Gallery content.

To enable TLS 1.2 for the current PowerShell prompt, run the following command:

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProto-
colType]::Tls12
```

To fix this issue permanently on a computer, you need to create registry keys. You can run the following two commands to create the necessary keys:

```
Set-ItemProperty -Path 'HKLM:\SOFTWARE\Wow6432Node\Microsoft\.NetFramework\
v4.0.30319'-Name 'SchUseStrongCrypto' -Value '1' -Type DWord


Set-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\.NetFramework\v4.0.30319'
-Name 'SchUseStrongCrypto' -Value '1' -Type DWord
```

Private PowerShell Gallery

You can implement a private PowerShell gallery for your organization by creating your own NuGet feed. You can create a NuGet feed 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 specifying the source location. After the repository is registered, users can search it, just like the PowerShell Gallery.

Additional reading: For more information about creating a NuGet feed, refer to **Hosting your own NuGet feeds**[3].

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 file or selecting it and then selecting Enter to run it, but that process doesn't 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 or selecting it and then selecting Enter. This is

2 https://aka.ms/ue14hl
3 https://aka.ms/vm0ys1

especially problematic when the file extension is hidden, and malware is included as an email attachment. For example, an attached file named **receipt.txt.vbs** would display as **receipt.txt** and users would run it accidentally, thinking it's a simple text file. This isn't a concern for Windows PowerShell scripts because of the actions required to run a script.

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 or select it and then select Enter, it opens in Notepad. This means that users can’t be tricked into running a Windows PowerShell script by double-clicking it or selecting it and then selecting Enter.

When you right-click a Windows PowerShell script or activate its context menu, 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 doesn't 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's already open.

Running scripts at the PowerShell prompt

When you run an executable file at a command prompt, you can enter its name to run it 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 can't use this process to run Windows PowerShell scripts, because it doesn't 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're digitally signed.
- **RemoteSigned**. Scripts that are downloaded can only be run if they're digitally signed.
- **Unrestricted**. All scripts can be run, but a confirmation prompt displays when running unsigned scripts that are downloaded.
- **Bypass**. All scripts are run without prompts.

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 the **Computer Configuration\Policies\Administrative Templates\Windows Components\Windows PowerShell\Turn on Script Execution** Group Policy setting to override the local setting.

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've modified a script downloaded from the internet, the script still has 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'll learn how to set the script execution policy.

Demonstration steps

1. On **LON-CL1**, open File Explorer, and then browse to **E:\Mod07\Democode**.
2. Rename **HelloWorld.txt** to **HelloWorld.ps1**.
3. Verify what happens when you double-click **HelloWorld.ps1** or select it and then select Enter.
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, run the following command:

   Set-Location E:\Mod07\Democode

7. To verify that **HelloWorld.ps1** is in the current directory, at the Windows PowerShell prompt, run the following command:

   Get-ChildItem HelloWorld.ps1

8. To run **HelloWorld.ps1** by using the full path, at the Windows PowerShell prompt, run the following command:

   E:\Mod07\Democode\HelloWorld.ps1

9. To verify that you can't run **HelloWorld.ps1** without specifying a path, at the Windows PowerShell prompt, run the following command:

   HelloWorld.ps1

10. To run **HelloWorld.ps1** in the current directory, at the Windows PowerShell prompt, run the following command:

    ```
    .\HelloWorld.ps1
    ```

11. To review the current execution policy, at the Windows PowerShell prompt, run the following command:

    ```
    Get-ExecutionPolicy
    ```

12. To prevent all scripts from running, at the Windows PowerShell prompt, run the following command:

    ```
    Set-ExecutionPolicy Restricted
    ```

13. To verify that all scripts are blocked, at the Windows PowerShell prompt, run the following command:

    ```
    .\HelloWorld.ps1
    ```

14. To allow all scripts to be run, at the Windows PowerShell prompt, run the following command:

    ```
    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's not very flexible. When you set an execution policy, you can only check that the script was downloaded and that it's signed.

Another alternative for controlling the use of Windows PowerShell scripts is AppLocker. With AppLocker, you can set various restrictions that limit the running 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 was 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.

Additional reading: For more information about **ConstrainedLanguage** mode, refer to about_Language_Modes in the Windows PowerShell help or **About Language Modes**[4].

4 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 policy, any modified scripts must be updated with a new digital signature. This prevents administrators or other staff from making random script changes. For example, if your organization has a set of approved scripts for managing Active Directory Domain Services (AD DS) users, this configuration will prevent helpdesk 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's 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's trusted by the computers.

You add a digital signature by using the **Set-AuthenticodeSignature** cmdlet, as the following code depicts:

```
$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'll learn how to digitally sign a Windows PowerShell script.

Demonstration steps

Install a code-signing certificate

1. On **LON-CL1**, open a Microsoft Management Console (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 to 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.

Digitally sign a certificate

1. To review the code-signing certificates installed for the current user, at the Windows PowerShell prompt, run the following command:

```
Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
```

2. To place the code-signing certificate in a variable, at the Windows PowerShell prompt, run the following command:

   ```
   $cert = Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert
   ```

3. To digitally sign a script, at the Windows PowerShell prompt, run the following command:

   ```
   Set-AuthenticodeSignature -FilePath E:\Mod07\Democode\HelloWorld.ps1 -Certificate $cert
   ```

Review the digital signature

- Open **E:\Allfiles\Mod07\Democode\HelloWorld.ps1**, and then verify that a digital signature has been added.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

True or False? To run a script in the current directory, enter the name of the script and select Enter.

☐ True

☐ False

Question 2

Which execution policy prevents unsigned scripts from running if they were downloaded?

☐ `Restricted`

☐ `AllSigned`

☐ `RemoteSigned`

☐ `Unrestricted`

Script constructs

Lesson overview

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'll learn how to use scripting constructs.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the syntax of the **ForEach** construct.
- Use the **ForEach** construct.
- Describe the syntax of the **If** construct.
- Use the **If** construct.
- Describe the syntax of the **Switch** construct.
- Use the **Switch** construct.
- Describe how to use the **For** construct.
- Describe 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:

```
ForEach ($user in $users) {
     Set-ADUser $user -Department "Marketing"
}
```

In the previous example, there's an array named `$users` that contains Active Directory Domain Services (AD DS) user objects. The **ForEach** construct processes the Windows PowerShell commands between the braces once for each object. When the commands are being processed, `$user` is the variable that contains each item from the array. On the first iteration, `$user` contains `$users[0]`, and on the second iteration, `$user` contains `$user[1]`. This continues until all items in the array have been processed once.

In a script, the **ForEach** construct is the most common way to process items that you've placed into an array. It's easy to use because you don't need to know the number of items to process them.

The previous example 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 review. The indent is not a technical requirement, but it's a good practice.

The 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 notice examples of variables with a single letter that's the same as the initial letter of the array. However, this should only be used in simple code where it's easy to tell that they're related.

Parallel performance

In PowerShell 7, the *-Parallel* parameter was added to the **ForEach-Object** cmdlet. This allows the pipeline to process multiple objects simultaneously. Processing multiple objects simultaneously can provide better performance than a standard **ForEach** loop. You should consider this if you're using PowerShell 7. The following example explains how to use the **ForEach-Object** with the *-Parallel* parameter.

```
$users | ForEach-Object -Parallel { Set-ADUser $user -Department "Market-
ing" }
```

By default, the *-Parallel* parameter allows five items to be processed at a time. You can modify this to be larger or smaller by using the *-ThrottleLimit* parameter.

Demonstration: Using a ForEach loop

In this demonstration, you'll learn how to use the **ForEach** construct.

Demonstration steps

1. On **LON-CL1**, open File Explorer, and then browse to **E:\Mod07\Democode**.
2. Rename **AZ-040_Mod07_Demo3.txt** to **AZ-040_Mod07_Demo3.ps1**.
3. Open **E:\Mod07\Democode\AZ-040_Mod07_Demo3.ps1** in Windows PowerShell ISE.
4. 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've 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:

```
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 previous example, the condition `$freeSpace -le 5GB` is evaluated first. If this condition is true, the script block in the braces is run, and no further processing happens in the **If** construct. In this case, a message indicating that there's less than 5 GB of free disk space is displayed.

If the first condition isn't true, the condition `$freeSpace -le 10GB` that's defined for **ElseIf** is evaluated. If this condition is true, the script block in the braces is run, and no further processing happens in the **If** construct. In this example, there's a single **ElseIf**, but you can have multiple **ElseIf** statements or none.

If all conditions aren't true, then the script block for **Else** is run. **Else** is optional.

Note: When you're 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'll learn how to use the **If** construct.

Demonstration steps

1. On **LON-CL1**, open File Explorer, and then browse to **E:\Mod07\Democode**.
2. Rename **AZ-040_Mod07_Demo4.txt** to **AZ-040_Mod07_Demo4.ps1**.
3. Open **E:\Mod07\Democode\AZ-040_Mod07_Demo4.ps1** in Windows PowerShell ISE.
4. Review the code, and then run the script.
5. Update the value of `$freeSpace` to **11GB**, and then run the script again.
6. Update the value of `$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 has a script block for each value. The script block for each value is run if that value matches the variable. There's also a **Default** section that runs only if there are no matches.

The **Switch** construct uses the following syntax:

```
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's 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 run. This is different from an **If** construct in which only one script block is run.

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 that it's 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 previous example, 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'll learn how to use the **Switch** construct.

Demonstration steps

1. On **LON-CL1**, open File Explorer, and then browse to **E:\Mod07\Democode**.
2. Rename **AZ-040_Mod07_Demo5.txt** to **AZ-040_Mod07_Demo5.ps1**.
3. Open **E:\Mod07\Democode\AZ-040_Mod07_Demo5.ps1** in Windows PowerShell ISE.
4. Review the code, and then run the script.
5. Update the value of `$computer` to **VAN-SRV1**, and then run the script again.
6. Update the value of `$computer` to **SEA-CL1**, and then run the script again.
7. 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:

```
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 previous example, 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 previous example, this loop is processed 10 times.

Note: When you're processing an array of objects, using the **ForEach** construct is preferred because you don't 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. These are **Do..While**, **Do..Until**, and **While**. All of these loop constructs 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 runs a script block until a specified condition isn't true. This construct guarantees that the script block is run at least once.

The **Do..While** construct uses the following syntax:

```
Do {
   Write-Host "Script block to process"
} While ($answer -eq "go")
```

Do..Until

The **Do..Until** construct runs a script block until a specified condition is true. This construct guarantees that the script block is run 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 runs a script block until a specified condition is false. It's similar to the **Do..While** construct, except that the script block isn't guaranteed to be run.

The **While** construct uses the following syntax:

```
While ($answer -eq "go") {
   Write-Host "Script block to process"
}
```

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're 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}
}
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which construct should you use to process an array when you're not sure how many items will be in the array?

- ☐ `If`
- ☐ `Do..While`
- ☐ `For`
- ☐ `ForEach`

Question 2

True or False? When you use the `Switch` *construct, multiple code blocks can be run.*

- ☐ True
- ☐ False

Import data from files

Lesson overview

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'll learn how to import data from a text file, CSV file, XML file, and JavaScript Object Notation (JSON) file.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe how to use **Get-Content** to review 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.
- 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 and each line in the text file becomes an item in the array.

The typical syntax for **Get-Content** is:

```
$computers = Get-Content C:\Scripts\computers.txt
```

The previous example 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's used to name the properties of each item in the array.

The **Import-Csv** cmdlet uses the following syntax:

```
$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 previous example, 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 by 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 doesn't 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's difficult to work with multivalued attributes, whereas XML can easily represent multivalued attributes or even objects that have other objects as a property.

The use of **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 to retrieve all the remaining objects.

Using ConvertFrom-Json

JavaScript Object Notation (JSON) is a lightweight data format that's similar to XML, because it can represent multiple layers of data. JSON is lightweight compared to XML because of its simpler syntax.

Windows PowerShell doesn't 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
```

Invoke-RestMethod

When you query a web service, the data is commonly provided using the JSON format. 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 in JSON format is automatically converted to objects. You don't 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 aren't 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'll learn 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, run the following command:

   Get-Content E:\Mod07\Democode\computers.txt

3. To place data from a text file into an array, at the Windows PowerShell prompt, run the following command:

   $computers = Get-Content E:\Mod07\Democode\computers.txt

4. To display the number of items in the `$computers` array, at the Windows PowerShell prompt, run the following command:

   $computers.count

5. To display the items in the `$computers` array, at the Windows PowerShell prompt, run the following command:

```
$computers
```

6. To import CSV data, at the Windows PowerShell prompt, run the following command:

   ```
   Import-Csv E:\Mod07\Democode\users.csv
   ```

7. To import CSV data into an array, at the Windows PowerShell prompt, run the following command:

   ```
   $users = Import-Csv E:\Mod07\Democode\users.csv
   ```

8. To display the count of items in the array, at the Windows PowerShell prompt, run the following command:

   ```
   $users.count
   ```

9. To display the first item in the array, at the Windows PowerShell prompt, run the following command:

   ```
   $users[0]
   ```

10. To display the property named **First** for the item, at the Windows PowerShell prompt, run the following command:

    ```
    $users[0].First
    ```

11. To import data from an XML file, at the Windows PowerShell prompt, run the following command:

    ```
    Import-Clixml E:\Mod07\Democode\users.xml
    ```

12. To import XML data into an array, at the Windows PowerShell prompt, run the following command:

    ```
    $usersXml = Import-Clixml E:\Mod07\Democode\users.xml
    ```

13. To review the number of items in the array, at the Windows PowerShell prompt, run the following command:

    ```
    $usersXml.count
    ```

14. To review the first item in the array, at the Windows PowerShell prompt, run the following command:

    ```
    $usersXml[0]
    ```

15. To review the properties for the items in the array, at the Windows PowerShell prompt, run the following command:

    ```
    $usersXml | Get-Member
    ```

16. Close the Windows PowerShell prompt.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which cmdlet imports data from a text file?

- ☐ `Get-Content`
- ☐ `Import-Csv`
- ☐ `Import-Clixml`
- ☐ `ConvertFrom-Json`

Question 2

True or False? You need to use `Invoked-RestMethod` *in combination with* `ConvertFrom-Json` *to place objects from a web API into a variable.*

- ☐ True
- ☐ False

Accept user input

Lesson overview

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 easier for others to use. In this lesson, you'll learn about multiple methods for accepting user input in a script.

Lesson objectives

After completing this lesson, you'll 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 create a script, it's usually 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 haven't signed in to the domain for 60 days. Another example is that 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'll likely find that you need to use variations of your scripts. For example, you might need to find computer accounts that haven't signed in for 30 days or find specific events in the event logs of servers other than your domain controllers.

In these examples, 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 beginning of the script where it's easily accessible, you make it easier to modify the script. However, this still requires modification of the script.

In an environment where many administrators share a common set of scripts, it's better not to modify scripts that'll go through an approval process. If scripts are digitally signed, each modification requires that the script be signed again. It's preferable to accept user input for values that change, rather than modify scripts.

Using Read-Host

You can use the **Read-Host** cmdlet to obtain input from users while a script is running. The request for user input could be a prompt to start the script or based on results from processing that's already happened in the script. For example, after performing a query for Active Directory Domain Services (AD DS) user objects and displaying the number of objects retrieved, the script could prompt a decision on whether to continue or stop. Alternatively, the script could request the specific event ID for which to search. The syntax for the **Read-Host** cmdlet is:

```
$answer = Read-Host "How many days"
```

The previous example stops processing the script and prompts the user with text as follows:

```
How many days:
```

At the prompt, the user enters a response and then selects 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's 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 example depicts:

```
Write-Host "How many days? " -NoNewline
$answer = Read-Host
```

Note: Input from **Read-Host** is limited to 1022 characters.

You can mask the input users enter at the prompt by using the *-MaskInput* or *-AsSecureString* parameters. Both parameters cause the characters entered by the user to display as asterisks (`*`). When *-MaskInput* is used, the response is collected as a String object. When *-AsSecureString* is used, the response is collected as a SecureString object. A SecureString object is required for scenarios such as setting passwords, where data shouldn't be stored as clear text in memory.

Using Get-Credential

As a best practice, administrators should have two user accounts. Each administrator should have a standard user account that's 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 you're still signed in to a standard user account.

Many scripts that administrators run require elevated privileges. For example, a script that creates Active Directory Domain Services (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're prompted for credentials, and all actions performed at that Windows PowerShell prompt use the credentials provided.

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** cmdlet is:

```
$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 *-User-Name* parameter.

Storing credentials by using Export-Clixml

You can store credentials to a file for later reuse without being prompted for credentials. To store credentials to a 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 a 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 helps keep the credentials secure, but it also means that they can't be shared with other users.

Storing credentials by using the SecretManagement module

Microsoft has released the **SecretManagement** module that you can use to store and retrieve credentials. This is a better method for storing credentials that can be shared among multiple users and computers. The cmdlets included in the **SecretManagement** module can access credentials from multiple secret vaults.

Some well-known vaults are:

- KeePass
- LastPass
- CredMan
- Azure KeyVault

The **SecretManagement** module is available in the PowerShell Gallery. You can install the **SecretManagement** module by running the following command:

```
Install-Module Microsoft.PowerShell.SecretManagement
```

Microsoft also provides the **SecretStore** module that you can use to create a local secret vault for storing credentials. However, similar to using **Export-Clixml**, the vault is stored on the local machine and in the current user context.

Using Out-GridView

Out-GridView is primarily used to review 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:

```
$selection = $users | Out-GridView -PassThru
```

In the previous example, 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 selects **OK**, the selected rows are stored in the `$selection` variable. You can 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 depicts the values that can be defined for the *-OutputMode* parameter.

Table 1: Values that can be defined for the -OutputMode parameter

Value	**Description**
None	This is the default value that doesn't 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 -PassThru parameter.

Note: Because users aren't forced to select a row in the **Out-GridView** window, you must ensure that your script properly handles the scenario where a row isn't selected.

Demonstration: Obtaining user input

In this demonstration, you'll learn how to obtain user input.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. To obtain user input by using **Read-Host**, at the Windows PowerShell prompt, run the following command. At the prompt, provide a number for the days:

   ```
   $days = Read-Host "Enter the number of days"
   ```

3. To review the data obtained by **Read-Host**, at the Windows PowerShell prompt, run the following command:

   ```
   $days
   ```

4. To obtain a credential, at the Windows PowerShell prompt, run the following command:

   ```
   $cred = Get-Credential
   ```

5. To display the credential information, at the Windows PowerShell prompt, run the following command:

   ```
   $cred | Format-List
   ```

6. To store the credential in a file, at the Windows PowerShell prompt, run the following command:

   ```
   $cred | Export-Clixml -Path E:\Mod07\Democode\cred.xml
   ```

7. To review the content of the file, at the Windows PowerShell prompt, run the following command:

   ```
   Get-Content E:\Mod07\Democode\cred.xml
   ```

8. To display a list of computer accounts in **Out-GridView**, at the Windows PowerShell prompt, run the following command:

   Get-ADComputer -Filter * | Out-GridView

9. To allow a single section in the **Out-GridView** window, at the Windows PowerShell prompt, run the following command:

   $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, run the following command:

    $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's consistent with how users provide input for cmdlets. The consistency makes it easier 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:

```
Param(
   [string]$ComputerName
   [int]$EventID
)
```

The variable names defined in the **Param()** block are also the names of the parameters. In the previous example, the script containing this **Param()** block has the *-ComputerName* and *-EventID* parameters that can be used. When you enter the parameter names for the script, you can use tab completion just as you can for cmdlet parameters. The syntax for running a script with parameters is:

```
.\GetEvent.ps1 -ComputerName LON-DC1 -EventID 5772
```

Note: Parameters are positional by default. If the parameter names aren't 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 don't 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's a best practice to define variable types in a **Param()** block. When you define the variable types, if a user enters a value that can't 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's an option that you want to be on or off. When the script is run, the parameter's presence sets the variable to `$true`. If the parameter isn't present, then the value for a variable is `$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 don't 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 doesn't provide a value for the parameter. This ensures that every necessary parameter has a value.

The following example depicts how to set a default value:

```
Param(
   [string]$ComputerName = "LON-DC1"
)
```

Requesting user input

You can also prompt for input if the user doesn't provide a parameter value. This ensures that the user provides a value for a parameter when there isn't a logical default value that you can specify.

The following example depicts how to prompt users for input:

```
Param(
   [int]$EventID = Read-Host "Enter event ID"
)
```

Note: You can configure additional advanced options for parameters in a script, such as making a parameter mandatory, by using the **Parameter()** attribute in the **Param()** block.

Additional reading: For more information about the **Parameter()** attribute, refer to **about_Functions_Advanced_Parameters**[5].

Demonstration: Obtaining user input by using parameters

In this demonstration, you'll learn how to obtain user input by using parameters.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. Rename **E:\Mod07\Democode\AZ-040_Mod07_Demo8.txt** to **AZ-040_Mod07_Demo8.ps1**.
3. Open Windows PowerShell Integrated Scripting Environment (ISE) and open **E:\Mod07\Democode\AZ-040_Mod07_Demo8.ps1**.
4. Review the code.

5 https://aka.ms/about-functions-advanced-parameters

5. To set the current directory, at the Windows PowerShell prompt, run the following command:

   ```
   Set-Location E:\Mod07\Democode
   ```

6. To pass values to the script by position, at the Windows PowerShell prompt, run the following command:

   ```
   .\AZ-040_Mod07_Demo8.ps1 LON-DC1 300
   ```

7. To pass values to the script by parameter name, at the Windows PowerShell prompt, run the following command:

   ```
   .\AZ-040_Mod07_Demo8.ps1 -EventID 300 -ComputerName LON-DC1
   ```

8. To review the results when no parameter data is provided, at the Windows PowerShell prompt, run the following command:

   ```
   .\AZ-040_Mod07_Demo8.ps1
   ```

9. In Windows PowerShell ISE, modify line 2 to ask for user input when a computer name isn't supplied.
10. Modify line 3 to set a default value of **300** when an event ID isn't specified.
11. Save the script.
12. To review the results when no parameter data is provided, at the Windows PowerShell prompt, run the following command:

    ```
    .\AZ-040_Mod07_Demo8.ps1
    ```

13. When prompted for a computer name, enter **LON-DC1**, and then select Enter.
14. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

To accept named parameters, what do you need to add to a script?

- ☐ A `Param()` block with parameter names defined
- ☐ A `Param()` block with variable names defined
- ☐ A `Parameter()` block with parameter names defined
- ☐ A function with variable names defined

Question 2

Which cmdlet can you use to collect a username and password from a user?

- ☐ `Out-GridView`
- ☐ `Read-Host`
- ☐ `Get-Credential`
- ☐ `Import-Clixml`

Troubleshooting and error handling

Lesson overview

As you develop more scripts, you'll 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 partially through running to query values for variables.

Lesson objectives

After completing this lesson, you'll 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.

Understanding error messages

When you run Windows PowerShell commands, you'll sometimes encounter error messages. Large 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 review the complete message, it provides a good description of why the error occurred. Errors can occur for reasons such as:

- You made a mistake while entering code.
- You queried an object that doesn't exist.
- You attempted to communicate with a computer that's offline.

When errors occur, they're stored in the `$Error` array. The most recent error is always at index zero. When a new error is generated, it's inserted at `$Error[0]`, and the index of other errors is increased by one. It can be useful to review errors in `$Error` whenever you need to review a previous error message. For example, if you clear the screen, you can use `$Error` to review the most recent error message.

Adding script output

When a script isn't 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's 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 isn't behaving as you expect it to behave, because a variable doesn't have a value that you expect.

If you want to make your troubleshooting text more easily identifiable, you can use the **Write-Warning** cmdlet instead of **Write-Host**. **Write-Warning** displays the text you specify in an alternate color.

If you want to slow down the running of a script to enable you to review 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're ready for it to continue, you can use **Read-Host**.

While you're 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.

Advanced script output

If you've configured your script as an advanced script by using **CmdletBinding()** in the **Param()** block, you can also use the cmdlets in the following table as part of your script for troubleshooting.

Table 1: Cmdlets for troubleshooting

Cmdlet	Description
Write-Verbose	Text specified by **Write-Verbose** is displayed only when you use the `-Verbose` parameter when running the script. The value of `$VerbosePreference` specifies the action to take after the **Write-Verbose** command. The default action is **SilentlyContinue**.
Write-Debug	Text specified by **Write-Debug** is displayed only when you use the `-Debug` parameter when running the script. The value of `$DebugPreference` specifies the action to take after the **Write-Debug** command. The default action is **SilentlyContinue**, which displays no information to screen. You need to change this action to **Continue** so that debug messages are displayed.

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 aren't 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 following example depicts how to set a breakpoint at a specific line of a script:

```
Set-PSBreakPoint -Script "MyScript.ps1" -Line 23
```

When you set a breakpoint based on a line, you need to be careful when you edit the script. As you edit the script, you might add or remove lines, and the breakpoint won't affect the same code that you initially intended.

The following example depicts 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 Read-
Write
```

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 aren't shared between multiple Windows PowerShell prompts and are removed when the prompt is closed.

PowerShell ISE

In the Windows PowerShell Integrated Scripting Environment (ISE), you can set breakpoints based on the 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.

Visual Studio Code

Microsoft Visual Studio Code allows you to set and use breakpoints with more advanced options than the PowerShell ISE. You can configure conditional breakpoints that are triggered when variables are outside of a range or match a specific value.

Information about variable contents is also easier to find in Visual Studio Code. After a breakpoint is triggered and you're in the debugger, there's a **variables** section that displays the contents of variable so that you don't need to interrogate them.

Demonstration: Troubleshooting a script

In this demonstration, you'll learn how to troubleshoot a script.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. Rename **E:\Mod07\Democode\AZ-040_Mod07_Demo9.txt** to **AZ-040_Mod07_Demo9.ps1**.
3. Open Windows PowerShell Integrated Scripting Environment (ISE) and open **E:\Mod07\Democode\AZ-040_Mod07_Demo9.ps1**.
4. Review the code.
5. To set the current directory, at the Windows PowerShell prompt, run the following command:

   Set-Location E:\Mod07\Democode

6. To review the script output, at the Windows PowerShell prompt, run the following command:

   .\AZ-040_Mod07_Demo9.ps1

7. To review the error, at the Windows PowerShell prompt, run the following command:

$Error[0]

8. To clear the `$Error` variable, at the Windows PowerShell prompt, run the following command:

   $Error.Clear()

9. To create a breakpoint, at the Windows PowerShell prompt, run the following command:

   Set-PSBreakPoint .\AZ-040_Mod07_Demo9.ps1 -Line 5

10. To run the script, at the Windows PowerShell prompt, run the following command:

    .\AZ-040_Mod07_Demo9.ps1

11. To review the value `$ComputerName`, at the Windows PowerShell prompt, run the following command:

    $ComputerName

12. To test the value `$ComputerName`, at the Windows PowerShell prompt, run the following command:

    $ComputerName -eq $null

13. To test the value `$ComputerName`, at the Windows PowerShell prompt, run the following command:

    $ComputerName -eq ""

14. To exit the debug prompt, at the Windows PowerShell prompt, run the following command:

    exit

15. In Windows PowerShell ISE, on line 5, change `$null` to `""` and save the script.
16. To review all breakpoints, at the Windows PowerShell prompt, run the following command:

    Get-PSBreakPoint

17. To remove all breakpoints, at the Windows PowerShell prompt, run the following command:

    Get-PSBreakPoint | Remove-PSBreakPoint

18. To run the script, at the Windows PowerShell prompt, run the following command:

    .\AZ-040_Mod07_Demo9.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's not possible to continue processing after the error and the command stops. A non-terminating error occurs when Windows PowerShell determines that it's possible to continue processing after the error. When a non-terminating error occurs, the script running or pipeline running 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** isn't available on the network, **Get-WmiObject** generates 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 decide 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, use the following syntax:

```
$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 manage the error, commands must use the **Stop** action. You can trap and manage only terminating errors.

-ErrorAction parameter

All Windows PowerShell commands have the *–ErrorAction* parameter. This parameter has the alias *–EA*. The parameter accepts the same values as `$ErrorActionPreference`, and the parameter overrides the variable for that command. If you expect an error to occur on a command, you use *–ErrorAction* to set that command's error action to **Stop**. Doing this lets you trap and manage the error for that command but leaves all other commands to use the action in `$ErrorActionPreference`. An example is:

```
Get-WmiObject -Class Win32_BIOS -ComputerName LON-SVR1,LON-DC1 -ErrorAction
Stop
```

The only time that you'll modify `$ErrorActionPreference` is when you expect an error outside of a Windows PowerShell command, such as when you're running 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's not a Windows PowerShell command, it doesn't have the *–ErrorAction* parameter. You would instead set `$ErrorActionPrefer-`

`ence` to **Stop** before running the method, and then set the variable back to **Continue** after you run the method.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which command will display the most recent Windows PowerShell error message?

- ☐ `$Error`
- ☐ `$Error[1]`
- ☐ `$Error.Clear()`
- ☐ `$Error[0]`

Question 2

How can you pause a script while it's running to interrogate variables and investigate why the script is failing? Choose two.

- ☐ Use the `Set-PSBreakPoint` cmdlet
- ☐ Use the `Break` command
- ☐ Use the `Continue` command
- ☐ Configure a conditional breakpoint in Microsoft Visual Studio Code

Functions and modules

Lesson overview

When you create many scripts, you'll have snippets of code that you want to reuse. You'll also have snippets of code that you want to reuse within the same script. Rather than having the same code display multiple times in a script, you can create a function that displays once in the script, but 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'll learn to create functions and modules.

Lesson objectives

After completing this lesson, you'll 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 **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
```

In the previous example, the value for the -*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 the following 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's 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.

Table 1: Scopes

Scope	**Description**
Global	The global scope is for the Windows PowerShell prompt. Variables set at the Windows PowerShell prompt can be reviewed in all the scripts started at that Windows PowerShell prompt. Variables created at a Windows PowerShell prompt don't exist in other Windows PowerShell prompts or in instances of the Windows PowerShell Integrated Scripting Environment (ISE).
Script	The script scope is for a single script. Variables set within a script can be reviewed by all the functions within that script. 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 review the value of the variable in the script, the value of the variable in the script scope is returned.
Function	The function scope is for a single function. Variables set within a function aren't shared with other functions or the script. 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. Then, there are two variables of the same name in two separate scopes.

Note: To avoid confusion, it's a best practice to avoid using the same variable names in different scopes.

In addition to reviewing 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's 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.

The 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 doesn't 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'll learn how to create a function.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. Rename **E:\Mod07\Democode\AZ-040_Mod07_Demo10.txt** to **AZ-040_Mod07_Demo10.ps1**.
3. Open Windows PowerShell Integrated Scripting Environment (ISE) and open **E:\Mod07\Democode\AZ-040_Mod07_Demo10.ps1**.
4. Review the code.
5. To set the prompt location, at the Windows PowerShell prompt, enter the following command, and then select Enter:

   Set-Location E:\Mod07\Democode

6. To review the size of a folder, at the Windows PowerShell prompt, enter the following command, and then select Enter:

   .\AZ-040_Mod07_Demo10.ps1 -Path "C:\Windows"

7. To review the size of a folder including subfolders, at the Windows PowerShell prompt, enter the following command, and then select Enter:

   .\AZ-040_Mod07_Demo10.ps1 -Path "C:\Program Files" -Recurse

8. In Windows PowerShell ISE, to create a function, add `Function Get-FolderSize {` as the first line and `}` as the last line.
9. To call the function, at the end of the file, add `Get-FolderSize -Path "C:\Program Files" -Recurse`.
10. To call the function a second time, at the end of the file, add `Get-FolderSize -Path C:\Windows`.
11. Save the script, and then run it to review the results.
12. Close Windows PowerShell ISE and the Windows PowerShell prompt.

Creating a module

You can create modules to store functions and share those functions among scripts. After you put your functions into modules, they're 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 the cmdlet naming convention. For example, you would use the verb-noun format.

Note: Functions in modules can include comment-based help that's discoverable by using **Get-Help**. To support this, you need to include the help information in each function.

In many cases, you already have your functions in a Windows PowerShell script file. To convert a script file containing only functions 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\Documents\WindowsPowerShell\Modules**
- **C:\Program Files\WindowsPowerShell\Modules**
- **C:\Windows\System32\WindowsPowerShell\1.0\Modules**

Windows PowerShell 7 includes the following additional paths:

- **C:\Program Files\PowerShell\Modules**
- **C:\Program Files\PowerShell\7\Modules**

Note: If you store modules in **C:\Users\UserID\Document\WindowsPowerShell\Modules**, they're only available to a single user.

Modules aren't 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'd place it in **C:\Program Files\WindowsPowerShell\Modules\AdatumFunctions**.

Demonstration: Creating a module from a function

In this demonstration, you'll learn how to create a module.

Demonstration steps

1. On **LON-CL1**, open a Windows PowerShell prompt.
2. To set the prompt location, at the Windows PowerShell prompt, run the following command:

   Set-Location E:\Mod07\Democode\

3. To copy and rename a script file, at the Windows PowerShell prompt, run the following command:

   Copy-Item .\AZ-040_Mod07_Demo10.ps1 .\FolderFunctions.psm1

4. Open Windows PowerShell Integrated Scripting Environment (ISE) and open **E:\Mod07\Democode\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, run the following command:

```
New-Item -Type Directory "C:\Program Files\WindowsPowerShell\Modules\FolderFunctions"
```

8. To copy the .psm1 file, at the Windows PowerShell prompt, run the following command:

```
Copy-Item .\FolderFunctions.psm1 "C:\Program Files\WindowsPowerShell\Modules\FolderFunctions"
```

9. To verify that the module is recognized, at the Windows PowerShell prompt, run the following command:

```
Get-Module -ListAvailable F*
```

10. To verify that the module isn't loaded, at the Windows PowerShell prompt, run the following command:

```
Get-Module
```

11. To use the function in the module, at the Windows PowerShell prompt, run the following command:

```
Get-FolderSize -Path C:\Windows
```

12. To verify that the module is loaded, at the Windows PowerShell prompt, run the following command:

```
Get-Module
```

13. Close the Windows PowerShell ISE and the Windows PowerShell prompt.

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:

```
. 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.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which path should you use to store a module named `OrgFunctions.psm1` *for Windows PowerShell?*

☐ `C:\Program Files\PowerShell\7\Modules`

☐ `C:\Program Files\PowerShell\Modules`

☐ `C:\Windows\System32\WindowsPowerShell\1.0\Modules\OrgFunctions`

☐ `C:\Program Files\WindowsPowerShell\OrgFunctions`

Question 2

True or False? Dot sourcing is the preferred method for maintaining a centralized repository of functions.

☐ True

☐ False

Module 07 lab and review

Lab: Using scripts with PowerShell

Scenario

You've started to develop Windows PowerShell scripts to simplify administration in your organization. There are multiple tasks to accomplish, and you'll create a Windows PowerShell script for each one.

Objectives

After completing this lab, you'll be able to:

- Digitally sign a script.
- Process an array by using ForEach.
- Process items by using If statements.
- Create user accounts based on a CSV file.
- Query disk information from remote computers.
- Update a script to use alternate credentials.

Estimated time: 150 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

True or False? When you set a variable value within a function, that variable is available after returning from the function.

☐ True

☐ False

Question 2

Which of the following options are true about creating Windows PowerShell scripts? (Select three.)

☐ You need to test scripts downloaded from the internet before you use them in production.

☐ If a script is well-documented in a blog, then you don't need to test it.

☐ It's often easier to modify an existing script than to create an entirely new script.

☐ You should only download scripts from the PowerShell Gallery.

☐ You should perform iterative testing while creating a script.

Question 3

You want to implement an approval process for Windows PowerShell scripts, where only tested and approved scripts are digitally signed. Which execution policy should you select to support this?

- ☐ `Restricted`
- ☐ `AllSigned`
- ☐ `RemoteSigned`
- ☐ `Unrestricted`

Question 4

Which command should you use to stop a `ForEach` *loop from processing the current item, but process all remaining items?*

- ☐ `Break`
- ☐ `Continue`
- ☐ `ByPass`
- ☐ `Quit`

Question 5

Which constructs guarantee that a code block is run at least once?

- ☐ `If`
- ☐ `Do..While`
- ☐ `Do..Until`
- ☐ `While`
- ☐ `Switch`

Question 6

True or False? When you use `Import-Csv` *and the data in the CSV file doesn't have a header row, you can specify property names by using the* `–Header` *parameter.*

- ☐ True
- ☐ False

Question 7

You're composing a script that creates new user accounts. When prompting for the user password, which command should you use?

- ☐ `$pass = Read-Host "Password?" -AsSecureString`
- ☐ `$pass = Read-Host "Password?"`
- ☐ `$pass = Read-Host "Password?" -MaskInput`
- ☐ `$pass = Get-Credential`

Question 8

You're trying to add verbose output to a script and the verbose output isn't displaying. What are two possible reasons for this?

- ☐ You forgot to add `CmdletBinding()` in the `Param()` block.
- ☐ You forgot to set the `$VerbosePreference` variable.
- ☐ You forgot to include the `–Verbose` switch when you ran the script.
- ☐ You set the `$verbosePreference` variable to `SilentlyContinue`.
- ☐ You forgot to add the `Break` command.

Answers

Question 1

True or False? To run a script in the current directory, enter the name of the script and select Enter.

☐ True

■ False

Explanation
False is the correct answer. To run a script in the current directory, you need to enter `.\` *before the name of the script to specify that you're running it from the current directory.*

Question 2

Which execution policy prevents unsigned scripts from running if they were downloaded?

☐ `Restricted`

☐ `AllSigned`

■ `RemoteSigned`

☐ `Unrestricted`

Explanation
`RemoteSigned` *is the correct answer. When the execution policy is configured as* `RemoteSigned`*, only downloaded scripts must be signed.*

Question 1

Which construct should you use to process an array when you're not sure how many items will be in the array?

☐ `If`

☐ `Do..While`

☐ `For`

■ `ForEach`

Explanation
`ForEach` *is the correct answer. A* `ForEach` *loop operates once for each item in an array. You don't need to know the number of items in the array.*

Question 2

True or False? When you use the `Switch` construct, multiple code blocks can be run.

■ True

☐ False

Explanation
True is the correct answer. When wildcard matching is performed with the `Switch` *construct, there can be multiple matches. The code block for each match is run.*

Question 1

Which cmdlet imports data from a text file?

■ `Get-Content`

☐ `Import-Csv`

☐ `Import-Clixml`

☐ `ConvertFrom-Json`

Explanation
`Get-Content` *is the correct answer. The* `Get-Content` *cmdlet imports data from a text file. Each line in the text file becomes an item in an array.*

Question 2

True or False? You need to use `Invoked-RestMethod` in combination with `ConvertFrom-Json` to place objects from a web API into a variable.

☐ True

■ False

Explanation
False is the correct answer. When you use `Invoke-RestMethod` *to query a web API, the data is automatically converted into objects.*

Question 1

To accept named parameters, what do you need to add to a script?

☐ A `Param()` block with parameter names defined

■ A `Param()` block with variable names defined

☐ A `Parameter()` block with parameter names defined

☐ A function with variable names defined

Explanation
A `Param()` *block with variable names defined is the correct answer. To accept named parameters, you add a* `Param()` *block to the beginning of the script. Inside the* `Param()` *block, you need to add variable names. The variable names correspond to the names of the parameters that will be accepted.*

Question 2

Which cmdlet can you use to collect a username and password from a user?

☐ `Out-GridView`

☐ `Read-Host`

■ `Get-Credential`

☐ `Import-Clixml`

Explanation
`Get-Credential` *is the correct answer. When you use* `Get-Credential`*, the user is prompted to enter a username and password that can be stored in a variable.*

Question 1

Which command will display the most recent Windows PowerShell error message?

☐ `$Error`

☐ `$Error[1]`

☐ `$Error.Clear()`

■ `$Error[0]`

Explanation
`$Error[0]` *is the correct answer. Errors are stored in the* `$Errors` *array. The most recent error is stored in index 0.*

Question 2

How can you pause a script while it's running to interrogate variables and investigate why the script is failing? Choose two.

■ Use the `Set-PSBreakPoint` cmdlet

☐ Use the `Break` command

☐ Use the `Continue` command

■ Configure a conditional breakpoint in Microsoft Visual Studio Code

Explanation
"Use the `Set-PSBreakPoint` *cmdlet" and "Configure a conditional breakpoint in Visual Studio Code" are the correct answers. These are both methods for setting a breakpoint, which pauses the script while it's running.*

Question 1

Which path should you use to store a module named `OrgFunctions.psm1` for Windows PowerShell?

☐ `C:\Program Files\PowerShell\7\Modules`

☐ `C:\Program Files\PowerShell\Modules`

■ `C:\Windows\System32\WindowsPowerShell\1.0\Modules\OrgFunctions`

☐ `C:\Program Files\WindowsPowerShell\OrgFunctions`

Explanation
`C:\Windows\System32\WindowsPowerShell\1.0\Modules\OrgFunctions` *is the correct answer. There are multiple folders that Windows PowerShell recognizes for module storage and* `C:\Windows\System32\WindowsPowerShell\1.0\Modules` *is one of them. Modules need to be stored in a subfolder with the same name as the module file.*

Question 2

True or False? Dot sourcing is the preferred method for maintaining a centralized repository of functions.

☐ True

■ False

Explanation
False is the correct answer. Modules are the preferred method for centrally maintaining a repository of functions.

Question 1

True or False? When you set a variable value within a function, that variable is available after returning from the function.

☐ True

■ False

Explanation
False is the correct answer. A variable that's set in the function scope isn't available in the script scope.

Question 2

Which of the following options are true about creating Windows PowerShell scripts? (Select three.)

■ You need to test scripts downloaded from the internet before you use them in production.

☐ If a script is well-documented in a blog, then you don't need to test it.

■ It's often easier to modify an existing script than to create an entirely new script.

☐ You should only download scripts from the PowerShell Gallery.

■ You should perform iterative testing while creating a script.

Explanation
The correct answers are "You need to test scripts downloaded from the internet before you use them in production.", "It's often easier to modify an existing script than to create an entirely new script.", and "You should perform iterative testing while creating a script."

Question 3

You want to implement an approval process for Windows PowerShell scripts, where only tested and approved scripts are digitally signed. Which execution policy should you select to support this?

☐ `Restricted`

■ `AllSigned`

☐ `RemoteSigned`

☐ `Unrestricted`

Explanation
`AllSigned` *is the correct answer. When the execution policy is configured as* `AllSigned`*, then scripts that aren't digitally signed can't be run.*

Question 4

Which command should you use to stop a `ForEach` loop from processing the current item, but process all remaining items?

☐ `Break`

■ `Continue`

☐ `ByPass`

☐ `Quit`

Explanation
`Continue` *is the correct answer. When* `Continue` *is used in a loop, it stops processing the current item or loop iteration and starts the next iteration of the loop.*

Question 5

Which constructs guarantee that a code block is run at least once?

☐ `If`

■ `Do..While`

■ `Do..Until`

☐ `While`

☐ `Switch`

Explanation
`Do..While` *and* `Do..Until` *are the correct answers. Both of these loops process the code block and then evaluate whether the condition has been met to process the code block again or stop.*

Question 6

True or False? When you use `Import-Csv` and the data in the CSV file doesn't have a header row, you can specify property names by using the `–Header` parameter.

■ True

☐ False

Explanation
True is the correct answer. You can use the `–Header` *parameter to specify the names that would typically be included in the header row of a CSV file.*

Question 7

You're composing a script that creates new user accounts. When prompting for the user password, which command should you use?

■ `$pass = Read-Host "Password?" -AsSecureString`

☐ `$pass = Read-Host "Password?"`

☐ `$pass = Read-Host "Password?" -MaskInput`

☐ `$pass = Get-Credential`

Explanation
`$pass = Read-Host "Password?" -AsSecureString` *is the correct answer. This prompts the user for a password, masks user input while they enter it, and stores the password as a SecureString object in the* `$pass` *variable.*

Question 8

You're trying to add verbose output to a script and the verbose output isn't displaying. What are two possible reasons for this?

■ You forgot to add `CmdletBinding()` in the `Param()` block.

☐ You forgot to set the `$VerbosePreference` variable.

■ You forgot to include the `–Verbose` switch when you ran the script.

☐ You set the `$verbosePreference` variable to `SilentlyContinue`.

☐ You forgot to add the `Break` command.

Explanation
"You forgot to add `CmdletBinding()` *in the* `Param()` *block." and "You forgot to include the* `–Verbose` *switch when you ran the script." are the correct answers. Both of these actions are required to display verbose output when you run a script.*

Module 8 Administering remote computers with Windows PowerShell

Use basic Windows PowerShell remoting

Lesson overview

You can run commands on one or hundreds of computers with a single PowerShell command. PowerShell supports remote computing by using various technologies, including Windows Management Instrumentation (WMI), remote procedure call (RPC), and Web Services for Management (WS-Management, or *WS-MAN*). Using the WS-Management protocol, Windows PowerShell remoting lets you run any Windows PowerShell command on one or more remote computers. For example, you can establish persistent connections, start interactive sessions, and run scripts on remote computers. Although remoting is a complex technology, after you understand the underlying concepts working with Windows PowerShell remoting is fairly straightforward. In this lesson, you'll learn how to use remoting to perform administration on remote computers.

Lesson objectives

After completing this lesson, you'll 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 manage and to route through firewalls. Windows operating systems implement the protocol by using the Windows Remote Management (WinRM) service. PowerShell supports WMI, WS-Management, and Secure Shell (SSH) remoting. In PowerShell 6, Remote Procedure Calls (RPC)-based communiation is not supported. In PowerShell 7 and newer, RPC is supported only in Windows.

You must enable remoting on the computers on which you want to receive incoming connections, although no configuration is necessary on computers that are initiating outgoing connections. PowerShell remoting is enabled by default for incoming connections on all currently supported versions of Windows Server . You can also enable it on any computer that's running Windows PowerShell 3.0 or newer.

Note: While remoting is enabled by default on Windows Server operating systems, it's not enabled by default on client operating systems, including Windows 10 and Windows 11.

Windows PowerShell remoting uses WinRM, which can manage communications for other applications. For example, on a default Windows Server 2016 or newer installation, WinRM manages communications for 64-bit Windows PowerShell, 32-bit Windows PowerShell, and two Server Manager components.

Remoting architecture

Remoting starts with the WinRM service. It registers one or more listeners, with each listener accepting incoming traffic through either HTTP or HTTPS. 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 doesn't 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. When traffic is directed to an endpoint, WinRM starts the associated application, hands off the incoming traffic, and then 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 runs the commands. The process is listed as **Wsmprovhost** in the remote computer's process list. Windows PowerShell would then run those commands and convert 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 translates 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.

Windows PowerShell remoting without configuration

Many Windows PowerShell cmdlets have the *ComputerName* parameter that enables you to collect data and change settings on one or more remote computers. These cmdlets use varying communication protocols and work on all Windows operating systems without any special configuration.

These cmdlets include:

- **Restart-Computer**
- **Test-Connection**
- **Clear-EventLog**
- **Get-EventLog**
- **Get-HotFix**
- **Get-Process**
- **Get-Service**
- **Set-Service**
- **Get-WinEvent**
- **Get-WmiObject**

Typically, cmdlets that support remoting without special configuration have the *ComputerName* parameter and don't have the *Session* parameter.

To find these cmdlets in your session, enter:

```
Get-Command | where { $_.parameters.keys -contains "ComputerName" -and
$_.parameters.keys -notcontains "Session"}
```

PowerShell remoting over SSH

PowerShell remoting normally uses WinRM for connection negotiation and data transport. SSH is now available for Linux and Windows platforms and allows true multiplatform PowerShell remoting.

WinRM provides a robust hosting model for PowerShell remote sessions. SSH-based remoting doesn't currently support remote endpoint configuration and Just Enough Administration (JEA).

SSH remoting offers basic PowerShell session remoting between Windows and Linux computers. SSH remoting creates a PowerShell host process on the target computer as an SSH subsystem. Microsoft plans to eventually implement a general hosting model similar to WinRM to support endpoint configuration and JEA.

Note: The **New-PSSession**, **Enter-PSSession**, and **Invoke-Command** cmdlets now have a new parameter set to support this new remoting connection.

To use PowerShell remoting over SSH, you must install PowerShell 6 or newer and SSH on all computers and then install both the SSH client (ssh.exe) and server (sshd.exe) executables so that you can remote to and from the computers. OpenSSH for Windows is available starting with Windows 10 build 1809 and Windows Server 2019. For Linux, install the version of SSH (including the sshd.exe server) that's appropriate for your platform. You also need to install the current version of PowerShell from GitHub to ensure that the SSH remoting feature is available. You should configure the SSH server to create an SSH subsystem to host a PowerShell process on the remote computer. You also need to enable either password or key-based authentication.

Remoting versus remote connectivity

Remoting is the name of a specific Windows PowerShell feature, not to be confused with the more generic concept of remote connectivity. Remoting is a generalized way to transmit any command to a remote computer so it runs locally on that computer. The command that you run doesn't have to be available on the computer that initiates the connection. Only the remote computers must be able to run it.

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 currently in use.

Many commands implement their own communications protocols, although in the future many of them might instead be changed to use remoting. For example, **Get-WmiObject** uses 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 communicate with a specific web service gateway by using their own protocol. All these other forms of communication might have unique firewall requirements and might require specific configurations to be in place to operate.

Remoting security

By default, the endpoints that Windows PowerShell creates only allow connections by members of a particular group. Starting with Windows Server 2016 and Windows 10, these groups include the Remote Management Users group and the local Administrators group. In an Active Directory Domain Services (AD DS) domain, the latter also includes members of the Domain Admins domain global group, since that group is a member of the local Administrators group on every domain-joined computer. Prior to Windows Server 2016 and Windows 10, by default, only members of the local Administrators group were allowed to use PowerShell remoting. It is, however, possible to change the defaults. Each endpoint does have a system access control list (SACL) that you can modify to control exactly who can connect to it.

PowerShell Remoting and WinRM listen on the following ports:

- HTTP: 5985
- HTTPS: 5986

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. The remote computer you are connecting to uses those credentials to impersonate you and perform the tasks that you have specified using those credentials. If you have enabled auditing, in addition to the tasks that you perform, the tasks that PowerShell remoting performs on your behalf will also be audited. In effect, remoting is security-transparent and doesn't change your environment's existing security. With remoting, you can perform all the same tasks that you would perform while physically located in front of the local computer.

Note: On private networks, the default Windows Firewall rule for PowerShell Remoting is to accept all connections. On public networks, the default Windows Firewall rule allows PowerShell Remoting connections only from within the same subnet. You must explicitly change that rule to open PowerShell Remoting to all connections on a public network.

Security risks and mutual authentication

Delegating your credentials to a remote computer involves some security risks. For example, if an attacker successfully impersonates a known remote computer, you could potentially transmit 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 authenticate yourself to the remote computer, and 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. When you connect between trusted domain computers, mutual authentication occurs automatically. When you connect to non-domain joined computers, you must provide another form of mutual authentication in the form of an SSL certificate and the HTTPS protocol that must be set up in advance. Another option is to turn off the requirement for mutual authentication by adding the remote computer to your local TrustedHosts list. Note however, that TrustedHosts uses Windows NT LAN Manager (NTLM) authentication, which doesn't 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.

Note: The NTLM authentication protocol cannot ensure the identity of the target server; it can only ensure that it already knows your password. Therefore, you should configure target servers to use SSL for PowerShell Remoting. Obtaining an SSL certificate issued by a trusted Certification Authority that the client trusts and assigning it to the target server enhances security of the NTLM-based authentication, helping validate both the user identity and server identity.

Computer name considerations

For AD DS-based authentication to work, PowerShell remoting must be able to search for and retrieve Active Directory Domain Services (AD DS) computer objects, which means that you need to refer to target computers by using their fully qualified domain names (FQDN). IP addresses or Domain Name System (DNS) aliases, for example, won't work because they don't provide remoting with the mutual authentication it needs. If you must refer to a computer by IP address or by a DNS alias, you must either connect using HTTPS, which means that 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 enumerates the computers for which mutual authentication isn't possible. Computers must be listed with the same name that you'll 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. While the TrustedHosts list makes it easier to connect to nondomain computers without having to set up HTTPS, it bypasses an important security measure. It allows you to send your credentials 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 not to be 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.

Note: As a best practice, you should 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 doesn't offer privacy or encryption of the content of your communication. 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 to satisfy organizational security requirements.

In a domain environment that uses the default Kerberos authentication protocol, credentials are sent in the form of encrypted Kerberos tickets that don't 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. As a result, even if you use Basic authentication, passwords are not transmitted in clear text. However, when you connect by using HTTP and Basic authentication to a computer that isn't configured for HTTPS, credentials (including passwords) will be transmitted in clear text. This can occur, for example, 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 are transmitted 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 one specifically designated for new computer provisioning. If you have to routinely connect to a nondomain computer, you should configure it to support HTTPS.

Enabling remoting

It's important to understand 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 for making sure that any local firewall will allow the outgoing traffic.

Manually enabling PowerShell remoting

PowerShell remoting is enabled by default on Windows Server platforms. You can enable PowerShell remoting on other supported Windows versions, and you can also re-enable remoting if it becomes disabled. To manually enable Windows PowerShell remoting on a computer, run the Windows PowerShell **Enable-PSRemoting** cmdlet. This is a persistent change that you can disable later by running **Disable-PSRemoting**. Note that this task requires the privileges granted to local Administrators group.

The **Enable-PSRemoting** cmdlet performs the following operations:

1. Runs the **Set-WSManQuickConfig** cmdlet, which in turn performs the following tasks:
   - Starts the WinRM service.
   - Sets the startup type on the WinRM service to Automatic.
   - Creates a listener to accept requests on any IP address.
   - Enables a firewall exception for WS-Management communications.
   - Creates the simple-name and long-name session endpoint configurations, if needed.
   - Enables all session configurations.
   - Changes the security descriptor of all session configurations to allow remote access.
2. Restarts the WinRM service to make the preceding changes effective.

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 Windows Firewall won't allow exceptions when you're connected to a Public network.

Note: The **Set-WSManQuickConfig** cmdlet doesn't affect remote endpoint configurations created by Windows PowerShell. It only affects endpoints created with PowerShell version 6 and newer. To enable and disable PowerShell remoting endpoints that are hosted by Windows PowerShell, run the **Enable-PS-Remoting** cmdlet from within a Windows PowerShell session.

Enabling remoting by using a GPO

Many organizations will prefer to centrally control Windows PowerShell remoting enablement and settings through GPOs. Microsoft supports this capability. You must set up various settings in a GPO to duplicate the steps taken by **Enable-PSRemoting**. To enable remoting by using Group Policy, you should configure the **Allow Remote Server Management Through WinRM** policy setting in the appropriate GPO. This setting also allows you to filter IP addresses from which remote connections can be initiated. In addition to configuring this policy, you should also configure appropriate firewall exceptions, as described earlier.

Using one-to-one remoting

One-to-one remoting resembles the SSH tool that's used on many UNIX and Linux computers in that you use a command prompt on the remote computer. While the immplementation of remoting is quite different from SSH, their use cases are fairly similar. In Windows PowerShell, you enter commands on your local computer, which then transmits them to the remote computer where they run. 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 isn't built on the Telnet protocol.

To start one-to-one Windows PowerShell remoting, run the **Enter-PSSession** command, combined with its *–ComputerName* parameter. You can use other parameters to perform basic connection customization, which we will cover in later topics.

After you're connected, the Windows PowerShell prompt changes to indicate the computer to which you're connected. To exit the session and return to the local command prompt, run **Exit-PSSession**. If you close Windows PowerShell while connected, the connection will close on its own.

Using one-to-many remoting

One-to-many remoting lets you send a single command to multiple computers in parallel. Each computer will run the command that you transmit, serialize the results into XML, and transmit those results back to your computer. Your computer then deserializes the XML into objects and 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.

You can use one-to-many remoting using two different techniques:

- **Invoke-Command –ComputerName name1,name2 –ScriptBlock { command }**. This technique sends the command (or commands) contained in the script block to the computers that you list. 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 you list. The local computer opens the file and reads its contents. However, the remote computers don't 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

To help you manage the resources on your local computer, PowerShell includes a per-command throttling feature that lets you limit the number of concurrent remote connections established for each command. By default, Windows PowerShell will connect to only 32 computers at once. If you list more than 32 computers, the connections to additional computers will be queued. Once sessions to some of the computers from the first batch complete and return their results, connections to the computers in the next batch will be initiated.

You can alter this behavior by using the –*ThrottleLimit* parameter of **Invoke-Command**. Raising the number doesn't put an additional load on the remote computers. However, it does put an additional load on the computer where **Invoke-Command** was invoked. It also utilizes more bandwidth. Each concurrent connection is basically a thread of Windows PowerShell. Therefore, raising the number of computers consumes memory and processor speed on the local computer.

Passing values

The script block or file contents are transmitted as literal text to the remote computers that run them exactly as is. The computer doesn't parse the script block or file on which the **Invoke-Command** was run. Consider the following command example:

```
$var = 'BITS'
Invoke-Command –ScriptBlock { Get-Service –Name $var } –Computer LON-DC1
```

In this scenario, the variable `$var` is being set on the local computer rather than being included into the script block to be run on **LON-DC1**. In other words, `$var` is not defined or set in the PowerShell remoting session to **LON-DC1**. This is a common mistake that administrators new to Windows PowerShell often make.

Running commands locally and remotely

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 won't process any script block contents, but simply pass it to the remote computer. For example, consider the following 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.

Question: Will that command work when run on a remote computer? For example, 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 running commands remotely and locally, you can achieve a variety of useful goals.

Persistence

Using the techniques outlined here, every time you use **Invoke-Command**, the remote computer creates a new *wsmprovhost* process, and runs the command or commands. It then returns the results, and then closes that Windows PowerShell instance. Each successive **Invoke-Command**, even if made to the same computer, is akin to opening a new Windows PowerShell window. Any work done by a previous session won't exist unless you save it to a disk or some other persistent storage. For example, consider the following command:

```
Invoke-Command –ComputerName LON-DC1 –ScriptBlock { $x = 'BITS' }
Invoke-Command –ComputerName LON-DC1 –ScriptBlock { Get-Service –Name $x }
```

In this example, the **Get-Service** would fail, because it's dependent on the value of a variable created as part of the previous wsmprovhost process. When the first script invoked by the **Invoke-Command** completes, its variables are cleared from memory. To address this issue, you can create a wsmprovhost process on a remote computer so that you can successfully send successive commands to it.

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 that can be used to create such collections:

- **-ComputerName ONE,TWO,THREE**. A static, comma-separated list of computer names.
- **-ComputerName (Get-Content Names.txt)**. Reads names from a text file named **Names.txt**, assuming the file contains one computer name per line.
- **-ComputerName (Import-Csv Computers.csv | Select –ExpandProperty Computer)**. Reads a comma-separated value (CSV) file that's named **Computers.csv** and contains a column named **Computer** that contains computer names.
- **-ComputerName (Get-ADComputer –Filter * | Select –ExpandProperty Name)**. Queries every computer object in AD DS, which can take a significant amount of time in a large domain.

Common mistakes when using computer names

Be careful where you specify a computer name. For example, review the following command:

```
Invoke-Command –ScriptBlock { Get-Service –ComputerName ONE,TWO }
```

This command doesn't provide a *–ComputerName* parameter to **Invoke-Command**. Therefore, the command runs on the local computer. The local computer will run **Get-Service** targeting computers named **ONE** and **TWO**. The protocols used by **Get-Service** will be used instead of Windows PowerShell remoting. Compare this 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.

For more interactive Windows PowerShell remoting situations, you can manage individual sessions as separate entities. To do this, you first create a session by using the **New-PSSession** command. The benefit of using the **New-PSSession** command is that 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. You then can reference the variable later by using the **Invoke-Command** command. When finished, you can close persistent sessions by using the **Remove-PSSession** command.

Demonstration: Enabling and using remoting

In this demonstration, you'll learn how to:

- Enable remoting on client computer.
- Use remoting.

Demonstration steps

1. On **LON-CL1**, select the **Start** menu, right-click the Windows PowerShell tile or activate its context menu, and then select **Run as Administrator**.
2. To ensure that you have the correct execution policy in place, in the Windows PowerShell command window, enter **Set-ExecutionPolicy RemoteSigned**, and then press the Enter key:
3. In the **Execution Policy Change** dialog box, select **Yes**.
4. Enter **Enable-PSRemoting**, and then press the Enter key.

   Note: If you receive an error about a network connection being Public, point out the error to students, and explain that this is a common error. Then run the **Enable-PSRemoting -SkipNetworkProfileCheck** command.
5. Confirm all subsequent dialog boxes by selecting **Yes** or selecting the **Y** key.
6. Enter **Enter-PSSession –ComputerName LON-DC1**, and then press the Enter key.
7. Enter **Get-Process**, and then press the Enter key.
8. Enter **Exit-PSSession**, and then press the Enter key.
9. Enter the following command, and then press the Enter key:

   Invoke-Command –ComputerName LON-CL1,LON-DC1 –ScriptBlock { Get-EventLog –LogName Security –Newest 10 }

10. Leave the Windows PowerShell command window open for the next demonstration.

Remoting output versus local output

When you run a command such as **Get-Process** on your local computer, the command returns object or objects of the type System.Diagnostics.Process and adds them to the Windows PowerShell pipeline. These objects have properties, methods, and frequently, events. Methods provide the ability to perform task. For example, the **Kill()** method of a Process object terminates the process that this object represents.

Note: 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.

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 process can use only static information about an object—in other words, its properties.

When your computer receives the XML, it's 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's now of the type **Deserialized.System.Diagnostics.Process**, a related, but different, kind of object. The deserialized object has no methods and no events.

Given the serialization and deserialization which is part of PowerShell remoting, you should consider any objects that are obtained in this manner to be a static snapshot. The values of object properties are not updatable, and the objects cannot be used to perform any actions. Therefore, any tasks that require interacting with remote objects should be performed on the remote computer as part of the PowerShell remoting session.

For example, here is an example of a command that will not yield the desired results:

```
Invoke-Command –Computer LON-DC1 –ScriptBlock { Get-Process –Name Note* } |
Stop-Process
```

In this example, you're retrieving Process objects, but the task of stopping processes takes place on the local computer rather than the remote one. This will result in stopping any local processes that happen to have the names matching the remote ones.

The proper way to accomplish the intended outcome would be to run:

```
Invoke-Command –Computer LON-DC1 –ScriptBlock { Get-Process –Name Note* |
Stop-Process }
```

In this case, 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.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Why would an administrator decide to use remoting instead of managing a computer directly?

Question 2

What are some security concerns with remoting?

Use advanced Windows PowerShell remoting techniques

Lesson overview

Windows PowerShell remoting includes several advanced techniques that help achieve specific goals or alleviate specific shortcomings. In the previous lesson, you reviewed the shortcomings of some of the basic techniques. In this lesson, you'll learn about some useful advanced techniques that will help overcome these challenges.

Lesson objectives

After completing this lesson, you'll be able to:

- Configure common remoting options.
- Send parameters and local variables to remote computers.
- Describe the use of Windows PowerShell scopes.
- Send local variables to a remote computer.
- Configure multi-hop remoting authentication.

Demonstration: Sending local variables to a remote computer

In this demonstration, you'll learn two methods for passing local information to a remote computer by using **Invoke-Command**.

Demonstration steps

1. To demonstrate how the data in a variable might be provided by a user, in the Windows PowerShell command window, enter the following command, and then press the Enter key:

   ```
   $quantity = Read-Host "Query how many log entries?"
   ```

2. When you're prompted for a number of log entries that you want to review, enter any desired value (for example, 5), and then press the Enter key.
3. Enter the following command, and then press the Enter key:

   ```
   Invoke-Command –ArgumentList $quantity –ComputerName LON-DC1 –ScriptBlock { Param($x)
   Get-EventLog –LogName Security –newest $x }
   ```

   Point out to students how you can review the number of entries you specified for the Security log.
4. Now try the Using: scope modifier. Enter the following command, and then press the Enter key:

   ```
   Invoke-Command -ComputerName lon-dc1 -ScriptBlock {Get-EventLog -LogName Security –
   Newest $Using:quantity}
   ```

Point out to students that just as you did with *-ArgumentList* parameter you still review the number of entries you specified for the Security log, but the `$Using` scope modifier is easier to process.

5. Leave the Windows PowerShell command window open for the next demonstration.

Multi-hop remoting

Another challenge with remoting is related to delegating credentials across multiple remote connections. By default, credentials can be delegated across only one connection, or *hop*. This restrictions delegation prevents the remote computer from further delegating your credentials, since this could introduce an extra security risk.

In general, this is the scenario we want to address:

1. You're signed in to **ServerA**.
2. From **ServerA**, you start a remote PowerShell session to connect to **ServerB**.
3. A command you run on **ServerB** via your PowerShell Remoting session attempts to access a resource on **ServerC**.
4. Access to the resource on **ServerC** is denied because the credentials you used to create the Power-Shell Remoting session are not passed from **ServerB** to **ServerC**.

The need to perform multiple hop (or, *multi-hop*) delegation can often occur in production environments. For example, in some organizations administrators aren't permitted to connect directly from their client computers to a server in the datacenter. 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 doesn't permit this approach. After you're connected to a remote computer, your credential can go no further than the remote computer. Trying to access any resource that isn't located on that comput-er typically results in a failure because your access isn't accompanied by a credential. The solution is to enable Credential Security Support Provider (CredSSP).

Enabling CredSSP

CredSSP caches credentials on the remote server (**ServerB**, from the previous example). Because of this, you should be aware that using CredSSP opens you up to potential credential theft attacks. If the remote computer is compromised, the attacker has access to the user's credentials. CredSSP is disabled by default on both client and server computers. You should enable CredSSP only in the most trusted envi-ronments. For example, a domain administrator connecting to a domain controller could have CredSSP enabled because the domain controller is highly trusted.

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.

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. However, using the asterisk (`*`) wildcard by itself is too permissive because you would be enabling any computer to redelegate your credential, even an unau-thorized user. Instead, consider a limited wildcard pattern, such as *.ADATUM.com, which would limit redelegation to computers in that domain.

To configure the server, run **Enable-WsManCredSSP –Role Server**. No delegated computer list is required on the server. You also can configure these settings through Group Policy, which offers a more centralized and consistent configuration across an enterprise.

Note: There have been numerous security breaches documented while using CredSSP, and therefore it's no longer a preferred option. You should instead use constrained delegation.

Resource-based, Kerberos-constrained delegation

Starting with Windows Server 2012, you can forgo using CredSSP and instead use constrained delegation. *Constrained delegation* implements delegation of service tickets by using security descriptors rather than an allow list of server names. This allows 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.

Constrained delegation requires:

- Access to a domain controller in the same domain as the host computer from which the Windows PowerShell remoting commands are being run.
- Access to a domain controller in the domain hosting the remote server you're trying to access from the intermediate remote server.

The code for setting up the permissions requires a computer running Windows Server with the Active Directory PowerShell Remote Server Administration Tools (RSAT). You can add 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's a negative cache entry. You'll need to clear the cache on **LON-SVR2** by using one of the following techniques:

- Run the command `klist purge -li 0x3e7`. This is the preferred and fastest method.
- Wait 15 minutes for the cache to clear automatically.
- Restart **LON-SVR2**.

To test constrained delegation, run the following code example:

```
$cred = Get-Credential Adatum\TestUser
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
}
```

Just enough administration

Just Enough Administration (JEA) is a security technology that enables delegated administration for anything managed by PowerShell. With JEA, you can:

- Reduce the number of administrators on your machines by using virtual accounts or group-managed service accounts to perform privileged actions on behalf of regular users.
- Limit what users can do by specifying which cmdlets, functions, and external commands they can run.
- Better understand what your users are doing by reviewing transcripts and logs that depict exactly which commands a user ran during their session.

Highly privileged accounts used to administer your servers pose a serious security risk. Should an attacker compromise one of these accounts, they could launch lateral attacks across your organization. Each compromised account gives an attacker access to even more accounts and resources, and puts them one step closer to stealing company secrets, launching a denial-of-service (DOS) attack, and more.

It's not always easy to remove administrative privileges, either. Consider the common scenario where the DNS role is installed on the same machine as your Active Directory domain controller. Your DNS administrators require local administrator privileges to fix issues with the DNS server. But to do so, you must make them members of the highly privileged Administrators security group. This approach effectively gives DNS Administrators the ability to gain control over your entire domain and access to all the resources on that machine.

JEA addresses this problem through the principle of least privilege. With JEA, you can configure a management endpoint for DNS administrators that gives them access only to the PowerShell commands they need to get their job done. This means you can provide the appropriate access to repair a poisoned DNS cache or restart the DNS server without unintentionally giving them rights to Active Directory, or to browse the file system, or run potentially dangerous scripts. Better yet, when the JEA session is configured to use temporary, privileged virtual accounts, your DNS administrators can connect to the server by using non-admin credentials and still run commands that typically require admin privileges. JEA enables you to remove users from widely privileged local or domain administrator roles and carefully control what they can do on each machine.

JEA is a feature included in PowerShell 5.0 and newer. For full functionality, you should install the latest version of PowerShell available for your system. PowerShell Remoting provides the foundation on which JEA is built. It's necessary to ensure PowerShell Remoting is enabled and properly secured before you can use JEA.

When creating a JEA endpoint, you need to define one or more role capabilities that describe what someone can do in a JEA session. A *role capability* is a PowerShell data file with the .psrc extension that lists all the cmdlets, functions, providers, and external programs that are made available to connecting users.

You can create a new PowerShell role capability file with the **New-PSRoleCapabilityFile** cmdlet. You should then edit the resulting role capability file to allow the commands required for the role. The PowerShell help documentation contains several examples of how you can configure the file.

Test your knowledge

Use the following question to check what you've learned in this lesson.

Question 1

Why might you configure remoting to use ports other than the default ports?

Use PSSessions

Lesson overview

In this module, you've learned that each remoting command you used created a connection, used it, and then closed it. However, as previously discussed, this approach doesn't provide the option to persist session data across remote connections. Using a persistent connection allows you to interactively query and manage a remote computer. In this lesson, you'll 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'll be able to:

- Explain the purpose of persistent connections.
- Create and 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.

Persistent connections

So far, when you were executing **Enter-PSSession** or **Invoke-Command**, Windows PowerShell was establishing a connection to a remote computer, running the commands that you specified, returning the results to Windows PowerShell, and then closing the connection. This approach offers no persistence of session data across connections because each connection is starting a separate PowerShell session.

Windows PowerShell can create 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 (RDS) session.

After making a PSSession to a remote computer, you run the desired commands within the session, but you leave the PSSession running. By doing this, you can run additional commands in the session.

Disconnected sessions

In Windows PowerShell 3.0 and newer, you can also manually disconnect from sessions. This allows you to close the session in which a PSSession was established, even shut down the local computer, without disrupting commands running in the PSSession on the remote computer. This is particularly useful for running commands that take a long 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, such as:

- Maximum session run time

- Maximum idle time
- Maximum number of incoming connections
- Maximum number of sessions per administrator

You can explore these configuration parameters by running **dir WSMan:\localhost\shell**, and change them in that same location. You also can control many of the settings through Group Policy.

Creating and using a PSSession

You use the **New-PSSession** command to create a persistent connection. The command contains many of the same parameters as **Invoke-Command**, including *-Credential*, *–Port*, and *–UseSSL*. This is because you're creating the identical kind of connection that **Invoke-Command** creates. However, instead of closing this connection immediately, you're leaving it running.

PSSessions do have an idle timeout, after which the remote computer will close them automatically. A closed PSSession differs from a disconnected PSSession, because closed PSSessions cannot be reconnected. In this case, you can only remove the PSSession, and then recreate 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.

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.

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 }
```

For example, the following command can use the `$dc` variable to open a PSSession to **LON-DC1** within a script or code block:

```
$dc = New-PSSession –ComputerName LON-DC1
```

The following command creates remote sessions on **Server01** and **Server02**, and the session objects are stored in the `$s` variable:

```
$s = New-PSSession -ComputerName Server01, Server02
```

Now that the sessions are established, you can run any command in them. And because the sessions are persistent, you can collect data from one command and use it in another command.

For example, the following command runs a **Get-HotFix** command in the sessions in the `$s` variable, and it saves the results in the `$h` variable:

```
Invoke-Command -Session $s {$h = Get-HotFix}
```

The `$h` variable is created in each of the sessions in `$s`, but it doesn't exist in the local session. Now you can use the data in the `$h` variable with other commands in the same session, and the results are displayed on the local computer. For example:

```
Invoke-Command -Session $s {$h | where {$_.InstalledBy -ne "NTAUTHORITY\
SYSTEM"}}
```

Demonstration: Using PSSessions

In this demonstration, you'll learn how to create and manage PSSessions.

Demonstration steps

Perform the demonstration steps on the **LON-CL1** virtual machine in the Windows PowerShell console application.

1. On the **LON-CL1** virtual machine, select the **Start** menu, right-click the Windows PowerShell tile or activate its context menu, and then select **Run as Administrator**.
2. In the Windows PowerShell command window, enter `$dc = New-PSSession –ComputerName LON-DC1`, and then press the Enter key.
3. Enter `$all = New-PSSession –ComputerName LON-DC1,LON-CL1`, and then press the Enter key.
4. Enter **Get-PSSession**, and then press the Enter key.
5. Enter `$dc`, and then press the Enter key.
6. Enter **Enter-PSSession –Session $dc**, and then press the Enter key.
7. Enter **Get-Process**, and then press the Enter key.
8. Enter **Exit-PSSession**, and then press the Enter key.
9. Enter `$dc`, and then press the Enter key.
10. Enter the following command, and then press the Enter key:

    Invoke-Command –Session $all –ScriptBlock { Get-Service | Where { $_.Status –eq 'Running' }}

11. Enter `$dc | Remove-PSSession`, and then press the Enter key.
12. Enter **Get-PSSession**, and then press the Enter key.
13. Enter **Get-PSSession | Remove-PSSession**, and then press the Enter key.
14. Leave the Windows PowerShell command window open for the next demonstration.

Disconnected sessions

As you've 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 won't disconnect from the sessions, it will just 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 review or reconnect to another user's PSSessions on a computer.

Demonstration: Working with disconnected sessions

In this demonstration, you'll learn how to:

- Create a PSSession.
- Disconnect a PSSession.
- Display PSSessions.
- Reconnect to a PSSession.

Demonstration steps

Perform the demonstration steps on the **LON-CL1** virtual machine in the Windows PowerShell console application.

1. In the Windows PowerShell command window, to create a variable named `$dc`, which creates a PSSession, enter the following command, and then press the Enter key:

   ```
   $dc = New-PSSession –ComputerName LON-DC1
   ```

2. To disconnect from the PSSession, enter the following command, and then press the Enter key:

   ```
   Disconnect-PSSession –Session $dc
   ```

3. To open the disconnected PSSession, enter the following command, and then press the Enter key:

   ```
   Get-PSSession –ComputerName LON-DC1
   ```

4. To reconnect to the PSSession, enter the following command, and then press the Enter key:

   ```
   Get-PSSession –ComputerName LON-DC1 | Connect-PSSession
   ```

5. To confirm that the PSSession is available, enter `$dc`, and then press the Enter key.
6. To close the PSSession, enter **Remove-PSSession –Session $dc**, and then press the Enter key.

Implicit remoting

One of the ongoing problems in the Windows management space is version mismatch. For example, Windows Server 2019 and 2022 include a number of new Windows PowerShell commands. You can make these commands available on Windows 10 or Windows 11 as part of the Remote Server Administration Tools (RSAT). However, you might not be able to use the same approach with older versions of Windows.

If you have recently had to rebuild a workstation, you're familiar with another ongoing problem, which is the sheer amount of time it can take to track down and install administrative tools and Microsoft Management Consoles (MMCs) on a computer. Assuming all of them are compatible with your Windows versions, 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 isn't 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 (GUI), was in part to make servers perform better and need fewer updates. However, this also means that they can't run GUI tools and MMCs.

Implicit remoting brings tools to you

Implicit remoting brings a copy of a server's Windows PowerShell tools to your local computer. In reality, you're not copying the commands at all; you're 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's exactly as if you ran everything through **Invoke-Command**, but it's much more convenient. Commands also run quicker, because commands on the server are co-located with the server's functionality and data.

Using implicit remoting

While implicit remoting became available in Windows PowerShell 2.0, it became much easier to use starting with Windows PowerShell 3.0. All that's 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.

With implicit remoting, you have the option of adding a prefix to the noun of commands that you import. Doing this can make it easier, for example, to have multiple versions of the same commands loaded simultaneously without causing a naming collision. For example, if you import both Microsoft Exchange Server 2016 and Exchange Server 2019 commands, you might add the 2016 and 2019 prefix to each of them, 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.

The **Help** option 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.

Test your knowledge

Use the following question to check what you've learned in this lesson.

Question 1

What are some potential operational concerns for PSSessions?

Module 08 lab and review

Lab: Performing remote administration with PowerShell

Scenario

You're an administrator for Adatum Corporation and must perform maintenance tasks on a server running Windows Server 2019. You don't have physical access to the server, and instead plan to perform the tasks using Windows PowerShell remoting. You also have some tasks to perform on both a server and another client computer that runs Windows 10. In your environment, communication protocols such as remote procedure call (RPC) are blocked between your local computer and the servers. You plan to use Windows PowerShell remoting, and want to use sessions to provide persistence and reduce the setup and cleanup overhead that improvised remoting connections will impose.

Objectives

After completing this lab, you'll be able to:

- Enable remoting on a client computer.
- Run a task on a remote computer by using one-to-one remoting.
- Run a task on two computers by using one-to-many remoting.
- Create and manage PSSessions.
- Send commands to multiple computers in parallel.

Estimated time: 60 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

What is the main service that Windows PowerShell remoting uses?

Question 2

If PowerShell remoting is disabled, which command should you use to enable it?

Question 3

Which authentication method is recommended for multi-hop remoting?

Question 4

Which command should you use to create a persistent connection for PowerShell remoting?

Answers

Question 1

Why would an administrator decide to use remoting instead of managing a computer directly?

A computer might not always be physically available. Computers in geographically distant locations, for example, might be more easily managed remotely.

Question 2

What are some security concerns with remoting?

Remoting requires that credentials be delegated across the network, and it offers expanded reach and capability for administrators. Both capabilities can cause security concerns. However, remoting offers several features that enable organizations to help secure, monitor, and audit it. Remoting doesn't give administrators additional permissions. Instead, it gives them a more efficient way to exercise the permissions they already have.

Question 1

Why might you configure remoting to use ports other than the default ports?

In most cases, you wouldn't do this. The best reason to configure remoting to use different ports is if your organization uses an application that has to use the same ports.

Question 1

What are some potential operational concerns for PSSessions?

Because PSSessions are persistent, one concern is that multiple administrators might open several PSSessions to a single server. This could potentially create a large amount of processing and memory overhead on the server. You can lessen this concern by configuring remoting options appropriately to limit the number of PSSessions one administrator can create. You can also limit the total number of administrators who might create concurrent PSSessions on a server. The default quota limit should be sufficient in most cases. In the `WSMan` properties, `MaxShellsPerUser` is set to `5` by default.

Question 1

What is the main service that Windows PowerShell remoting uses?

Windows PowerShell remoting uses Windows Remote Management (WinRM), which can manage communications for other applications.

Question 2

If PowerShell remoting is disabled, which command should you use to enable it?

You should use the Enable-PSRemoting command.

Question 3

Which authentication method is recommended for multi-hop remoting?

When using Windows Server 2012 or newer operating systems, you should use Kerberos-constrained delegation.

Question 4

Which command should you use to create a persistent connection for PowerShell remoting?

You should use New-PSSession command.

Module 9 Managing Azure resources with PowerShell

Azure PowerShell

Lesson overview

You can manage Microsoft Azure resources by using the Azure portal, which is usually the most common way of administration. However, for some tasks, Azure PowerShell is more convenient. In this lesson, you'll learn about the Azure PowerShell environment and the Az module for Windows PowerShell. Also, you'll learn about ways to manage Azure Active Directory (Azure AD) by using PowerShell modules.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe Azure PowerShell.
- Describe the Azure Az PowerShell module.
- Install the Azure Az PowerShell module.
- Migrate Azure PowerShell from AzureRM to Az.
- Describe the Azure Active Directory Module for Windows PowerShell and Azure Active Directory PowerShell for Graph modules.

Azure PowerShell overview

Azure PowerShell is a module that you add to Windows PowerShell or PowerShell Core to let you connect to your Azure subscription and manage resources. It provides a set of cmdlets that you can use to manage and administer Azure resources directly from the PowerShell command line. Azure PowerShell makes it easier to interact with Azure, but also provides powerful features for automation. Written in .NET Standard, Azure PowerShell works with PowerShell 5.1 on Windows, PowerShell 7.0.6 LTS, and PowerShell 7.1.3 or newer on all platforms.

Azure PowerShell requires PowerShell to function. PowerShell provides services such as the shell window and command parsing. The Azure PowerShell module adds the Azure-specific commands.

You should use Azure PowerShell when you want to build automated tools that use the Azure Resource Manager model. You can use it in your browser with Azure Cloud Shell, or install it on your local Windows, Mac, or Linux machine. In both cases, you have two modes to choose from. You can use Azure PowerShell in interactive mode, in which you manually issue one command at a time, or in scripting mode, where you run a script that consists of multiple commands.

What is the Azure Az PowerShell module?

The Az PowerShell module is a set of cmdlets for managing Azure resources directly from PowerShell. PowerShell provides powerful features for automation that you can use to manage your Azure resources; for example, in the context of a continuous integration and continuous delivery (CI/CD) pipeline.

The Az PowerShell module is the replacement for AzureRM and is the recommended version to use for interacting with Azure. To keep up with the latest Azure features in PowerShell, you should migrate to the Az PowerShell module.

Note: There's no support for having both the AzureRM and Az modules installed for PowerShell 5.1 on Windows at the same time.

Benefits of the Az PowerShell module

The Az PowerShell module features the following benefits:

- Security and stability:
  - Token cache encryption
  - Prevention of man-in-the-middle attack type
  - Support for authentication with Active Directory Federation Services (AD FS) in Windows Server 2019
  - Username and password authentication in PowerShell 7
  - Support for features such as continuous access evaluation
- Support for all Azure services:
  - All generally available Azure services have a corresponding supported PowerShell module
  - Multiple bug fixes and API version upgrades since AzureRM
- New capabilities:
  - Support in Cloud Shell and cross-platform
  - Ability to get and use access tokens to access Azure resources
  - Cmdlets for advanced Representational State Transfer (REST) operations with Azure resources

The Az PowerShell module is based on the .NET Standard library and works with PowerShell 7 and newer on all platforms including Windows, macOS, and Linux. It's also compatible with Windows PowerShell 5.1.

Note: PowerShell 7 and newer are the recommended versions of PowerShell for use with Az PowerShell on all platforms.

Az is the most current PowerShell module for Azure. You can log issues or feature requests directly on the GitHub repository. You can also contact Microsoft support if you have a support contract. Feature requests will be implemented in the latest version of Az. Critical issues will be implemented on the last two versions of Az.

Note: Because Az PowerShell modules now have all the capabilities of AzureRM PowerShell modules and more, Microsoft plans to retire AzureRM PowerShell modules on February 29, 2024.

Installing the Azure Az PowerShell module

The Azure Az PowerShell module is a rollup module. Installing it downloads the available Az PowerShell modules and makes their cmdlets available for use. The Azure Az PowerShell module works with PowerShell 7.x and newer versions on all platforms. Azure PowerShell has no additional requirements when you run it on PowerShell 7.x and newer versions.

To check your PowerShell version, run the following command from within a PowerShell session:

```
$PSVersionTable.PSVersion
```

Before installing the Azure Az PowerShell module, you should set your PowerShell script execution policy to **RemoteSigned**. You can do this by running the following command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Methods to install the Az PowerShell module

You can install the Azure Az PowerShell module by using one of the following methods:

- The **Install-Module** cmdlet
- Azure PowerShell MSI
- Az PowerShell Docker container

Note: The Azure Az PowerShell module is preinstalled in Azure Cloud Shell. You can use it directly from the browser, without installing anything locally on your machine. You'll learn more about Azure Cloud Shell in the next lesson.

The **Install-Module** cmdlet

Using the **Install-Module** cmdlet is the preferred installation method for the Azure Az PowerShell module. You should install this module for the current user only. This is the recommended installation scope. This method works the same on Windows, macOS, and Linux platforms. To install the Az module, run the following command from a local PowerShell session:

```
Install-Module -Name Az -Scope CurrentUser -Repository PSGallery -Force
```

Although PowerShell 7.x is the recommended version of PowerShell, and **Install-Module** is the recommended installation option, you can also install the Az module within PowerShell 5.1 environment on Windows. If you're on Windows 10 version 1607 or higher, you already have PowerShell 5.1 installed. You should also make sure that you have .NET Framework 4.7.2 or newer installed and the latest version of PowerShellGet. To install the latest version of the PowerShellGet module within PowerShell 5.1, run the following command:

```
Install-Module -Name PowerShellGet -Force
```

You can then install the Az module by using the same command you use in PowerShell 7.1.

Azure PowerShell MSI

In some environments, it isn't possible to connect to the PowerShell Gallery. In such situations, you can install the Az PowerShell module offline, by downloading the Azure PowerShell MSI package. Keep in mind that the MSI installer only works for PowerShell 5.1 on Windows.

To update any PowerShell module, you should use the same method used to install the module. For example, if you originally used **Install-Module**, then you should use **Update-Module** to get the latest version. If you originally used the MSI package, then you should download and install the new MSI package.

Az PowerShell Docker container

It's also possible to run Azure PowerShell inside a Docker image. Microsoft provides Docker images with Azure PowerShell preinstalled. The released images require Docker 17.05 or newer. The latest container image contains the latest version of PowerShell and the latest Azure PowerShell modules supported with the Az module.

To download the image and start an interactive PowerShell session, you should run the following commands:

```
docker pull mcr.microsoft.com/azure-powershell
docker run -it mcr.microsoft.com/azure-powershell pwsh
```

Starting to work with Azure PowerShell

To start working in the Azure PowerShell environment, you should first sign in with your Azure credentials. This step is different from working in pure PowerShell. Your Azure credentials are the same credentials you use to sign in to the Azure portal or other Azure-based resources.

To sign in to Azure from Azure PowerShell, run the following command:

```
Connect-AzAccount
```

After running this command, you'll be prompted to sign in with your Azure credentials. After you successfully authenticate to Azure, you can start using commands from the Az module to manage your Azure resources.

Migrating Azure PowerShell from AzureRM to Az

Scripts created for the AzureRM cmdlets won't automatically work with the Az module. To make the transition easier, the AzureRM to Az migration toolkit was developed. No migration to a new command set is ever convenient, but it's important that you understand how to transition to the Az PowerShell module.

The new cmdlet names have been designed to be easier to learn. Instead of using AzureRm or Azure in cmdlet names, you use Az cmdlets. For example, the old cmdlet **New-AzureRMVm** has become **New-AzVm**. However, migration is more than just becoming familiar with the new cmdlet names. There are renamed modules, parameters, and other important changes.

Before taking any migration steps, check which versions of AzureRM are installed on your system. Doing so allows you to make sure scripts are already running on the latest release and let you know which versions of AzureRM must be uninstalled.

To check which versions of AzureRM you've installed, run the following command:

```
Get-Module -Name AzureRM -ListAvailable -All
```

Note: The latest available release of AzureRM is 6.13.1. If you don't have this version installed, your existing scripts might need additional modifications to work with the Az module.

The recommended option to migrate from AzureRM to the Az PowerShell module is to use automatic migration. For this, you need to install the AzureRM to Az migration toolkit by running the following command:

```
Install-Module -Name Az.Tools.Migration
```

With the AzureRM to Az migration toolkit, you can generate a plan to determine what changes will be performed on your scripts before making any modifications to them and before installing the Az PowerShell module.

Additional reading: To learn about the steps for automatic migration, refer to **Quickstart: Automatically migrate PowerShell scripts from AzureRM to the Az PowerShell module**[1].

You can also use Microsoft Visual Studio Code to migrate your existing scripts. To do so, you first need to install the Azure PowerShell extension for Visual Studio Code. Then, you need to perform the following steps:

1. Load your AzureRM script in Visual Studio Code.
2. Open the command palette by selecting **Ctrl+Shift+P**.
3. Select the **Migrate Azure PowerShell** script.
4. Select the **AzureRM** source version.
5. Follow the recommended actions for each underlined command or parameter.

What are the Microsoft Azure Active Directory Module for Windows PowerShell and Azure Active Directory PowerShell for Graph modules?

The Azure Active Directory Module for Windows PowerShell provides cmdlets that you can use for Azure Active Directory (Azure AD) administrative tasks such as user management, domain management, and configuring single sign-on. This topic includes information about how to install these cmdlets for use with your directory.

You mostly need the Azure Active Directory Module for Windows PowerShell when you manage users, groups, and services such as Microsoft 365. However, Microsoft is replacing the Azure Active Directory Module for Windows PowerShell with Azure Active Directory PowerShell for Graph. The Azure Active Directory Module for Windows PowerShell cmdlets include **Msol** in their names, while the Azure Active Directory PowerShell for Graph cmdlets use **AzureAD** in their names.

1 https://aka.ms/quickstart-automatically-migrate-powershell-scripts-from-azurerm-to-the-az-powershell-module

Azure Active Directory Module for Windows PowerShell

The Azure Active Directory Module for Windows PowerShell is supported on the following Windows operating systems with the default version of Microsoft .NET Framework and Windows PowerShell:

- Windows 8.1
- Windows 8
- Windows 7
- Windows Server 2012 R2
- Windows Server 2012
- Windows Server 2008 R2

The easiest way to install the module is from the PowerShell Gallery. You can install the module with the **Install-Module** cmdlet by running the following command:

```
Install-Module MSOnline
```

Azure Active Directory PowerShell for Graph

Currently, the Azure Active Directory PowerShell for Graph module doesn't completely replace the functionality of the Azure Active Directory Module for Windows PowerShell module for user, group, and license administration. In some cases, you need to use both versions. You can safely install both versions on the same computer.

The Azure AD PowerShell for Graph module has two versions: a Public Preview version and a General Availability version. It isn't recommended to use the Public Preview version for production scenarios.

Additional reading: For more information about downloading either version of the Azure AD PowerShell for Graph module, refer to **Install Azure Active Directory PowerShell for Graph**[2].

To install the General Availability version of the Azure AD PowerShell for Graph module on your computer, run the following command:

```
Install-Module AzureAD
```

To install the public preview release of this module, run the following command:

```
Install-module AzureADPreview
```

Connecting to Azure AD with PowerShell

If you want to connect to the Azure AD service with the Azure Active Directory Module for Windows PowerShell, run the following command:

```
Connect-MsolService
```

If you use the Azure AD PowerShell for Graph module, and want to connect to Azure AD, run the following command:

2 https://aka.ms/install-azure-active-directory-powershell-for-graph

```
Connect-AzureAD
```

After running either of the previous commands, you'll be prompted for your Azure AD credentials. You should use the credentials that you use to sign in to Microsoft 365 or your Azure services. After you authenticate, you'll be able to use the cmdlets available for Azure AD management.

Additional reading: For more information about the Azure Active Directory PowerShell for Graph cmdlets, refer to **AzureAD**[3].

Additional reading: For more information about the Azure Active Directory Module for Windows PowerShell cmdlets, refer to **MSOnline**[4].

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Besides using Azure PowerShell in interactive mode, what's the other mode you can use it in to run commands?

Question 2

Is it possible to use the Az module for PowerShell on Mac computers?

Question 3

What's the preferred way to install the Az module for PowerShell? What version of PowerShell is recommended for this module?

3 https://aka.ms/azure-ad-2
4 https://aka.ms/msonline

Introduce Azure Cloud Shell

Lesson overview

Instead of using the locally installed PowerShell module for managing Azure resources, you could also use the Azure Cloud Shell environment. This option lets you use PowerShell or Bash environments and commands to manage Azure resources. Azure Cloud Shell is available in the Azure portal and also in the Microsoft 365 admin portal. In this lesson, you'll learn about Azure Cloud Shell and its features.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe Azure Cloud Shell.
- Describe features and tools for Azure Cloud Shell.
- Use Azure Cloud Shell.

Cloud Shell overview

The easiest way to get started with Azure PowerShell is by trying it out in an Azure Cloud Shell environment. Azure Cloud Shell is an interactive, browser-accessible shell for managing Azure resources. It provides the flexibility of choosing the shell experience that best suits the way you work. Linux users can opt for a Bash experience, while Windows users can opt for PowerShell.

Cloud Shell enables access to a browser-based, command-line experience built with Azure management tasks in mind. You can use Cloud Shell to work untethered from a local machine in a way only the cloud can provide.

The main characteristics of Azure Cloud Shell are that it:

- Is a temporary environment that requires a new or existing Azure file share to be mounted.
- Offers an integrated graphical text editor based on the open-source Monaco Editor.
- Authenticates automatically for instant access to your resources.
- Runs on a temporary host provided on a per-session, per-user basis.
- Times out after 20 minutes without interactive activity.
- Requires a resource group, storage account, and Azure file share.
- Uses the same Azure file share for both Bash and PowerShell.
- Is assigned to one machine per user account.
- Persists $HOME using a 5-GB image held in your file share.
- Has permissions that are set as a regular Linux user in Bash.

You can access the Cloud Shell in three ways:

- Direct link. Open a browser and refer to **https://shell.azure.com**.
- Azure portal. Select the Cloud Shell icon on the Azure portal.

- Code snippets. On **Microsoft Docs**[5] and **Microsoft Learn**[6], select the **Try It** option that displays with the Azure CLI and Azure PowerShell code snippets:

  az account show
  Get-AzSubscription

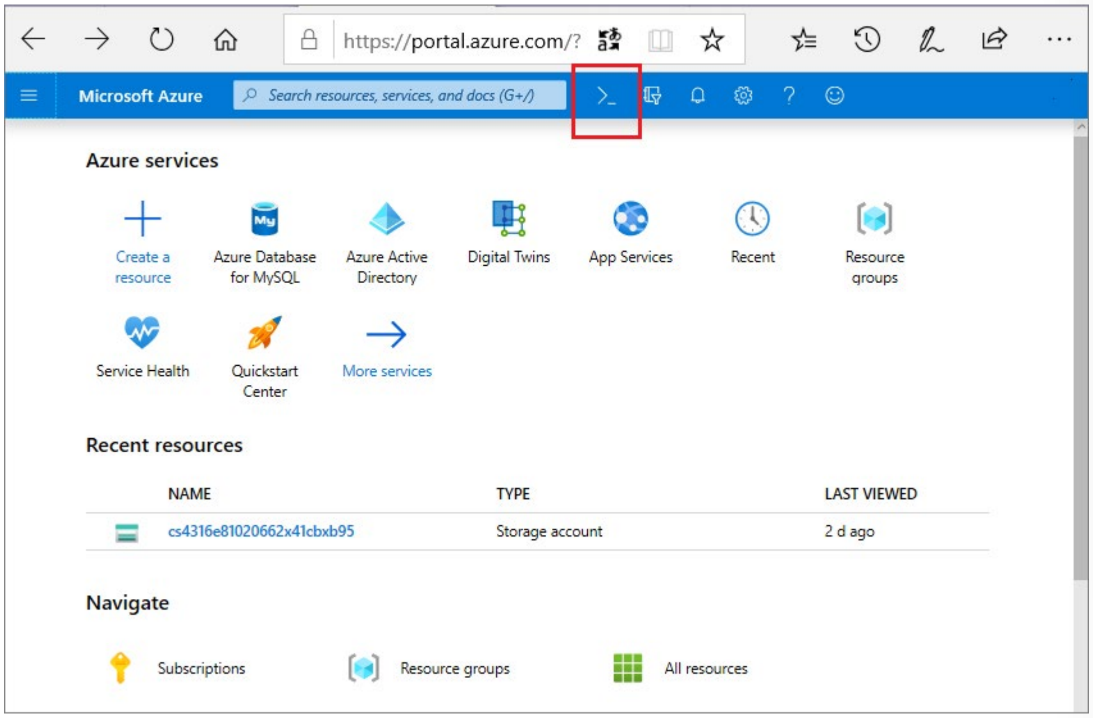
  The **Try It** option opens the Cloud Shell directly alongside the documentation using Bash (for Azure CLI snippets) or PowerShell (for Azure PowerShell snippets).

  To run the command:

  1. Use **Copy** in the code snippet.
  2. Use **Ctrl+Shift+V** (Windows/Linux) or **Cmd+Shift+V** (macOS) to paste the command.
  3. Select Enter.

Selecting your preferred shell experience

To choose between Bash or PowerShell, refer to the Azure portal and select the Cloud Shell icon, as the following screenshot depicts.

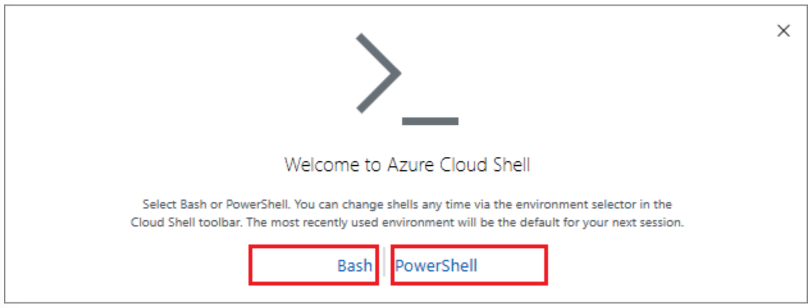


NAME	TYPE	LAST VIEWED
cs4316e81020662x41cbxb95	Storage account	2 d ago

Figure 1: Azure Cloud Shell icon

In the **Welcome to Azure Cloud Shell** dialog box, select **Bash** or **PowerShell**, as the following screenshot depicts.

5 https://aka.ms/microsoft-docs
6 https://aka.ms/microsoft-learn-2



Select Bash or PowerShell. You can change shells any time via the environment selector in the Cloud Shell toolbar. The most recently used environment will be the default for your next session.

Figure 2: Bash and PowerShell options

After the first launch, you can use the shell type drop-down list to switch between **Bash** and **PowerShell**.

Microsoft manages Azure Cloud Shell, so it comes with popular command-line tools and language support. Cloud Shell also helps securely authenticate automatically, so that you can instantly access your resources through the Azure CLI or Azure PowerShell cmdlets. Cloud Shell also offers an integrated graphical text editor based on the open-source Monaco Editor.

Cloud Shell machines are temporary, but your files are persisted in two ways: through a disk image, and through a mounted file share named **clouddrive**. On the first launch, Cloud Shell prompts to create a resource group, storage account, and Azure file share on your behalf. This is a one-time step and will be automatically attached for all sessions. A single file share can be mapped and will be used by both Bash and PowerShell in Cloud Shell.

Features and tools for Azure Cloud Shell

Cloud Shell offers a browser-accessible, preconfigured shell experience for managing Azure resources without the overhead of installing, versioning, and maintaining a machine yourself.

Cloud Shell provisions machines on a per-request basis, and as a result, machine state won't persist across sessions. Cloud Shell is built for interactive sessions, and therefore, shells automatically terminate after 20 minutes of shell inactivity.

Note: Azure Cloud Shell runs on Ubuntu 16.04 LTS.

Secure automatic authentication

Cloud Shell securely and automatically authenticates account access for the Azure CLI and Azure PowerShell. This helps you gain quick and more secure access to your resources.

$HOME persistence across sessions

To persist files across sessions, Cloud Shell moves through attaching an Azure file share on the first launch. When this completes, Cloud Shell will automatically attach your storage (mounted as `$HOME\clouddrive`) for all future sessions. Additionally, your `$HOME` directory is persisted as an .img in your Azure file share. Files outside of $HOME and machine state aren't persisted across sessions. Depending on the scenario, you should use recommended best practices when storing secrets such as SSH keys. Services such as Azure Key Vault have tutorials for setup.

Azure drive (Azure:)

PowerShell in Cloud Shell provides the Azure drive (Azure:). You can switch to the Azure drive with **cd Azure:** and back to your home directory with **cd ~**. The Azure drive enables easier discovery and navigation of Azure resources such as compute, network, and storage, similar to file system navigation. You can continue to use familiar Azure PowerShell cmdlets to manage these resources, regardless of the drive you're in. Any changes to the Azure resources, whether they're made directly in the Azure portal or by using Azure PowerShell cmdlets, are reflected in the Azure drive. You can run **dir -Force** to refresh your resources.

Manage Exchange Online

PowerShell in Cloud Shell contains a private build of the Exchange Online module. Run **Connect-EXOPSSession** to get your Exchange cmdlets. By using these cmdlets, you can manage your Exchange Online instance running in the Microsoft 365 environment.

Deep integration with open-source tooling

Cloud Shell includes preconfigured authentication for various open-source tools. The following table lists the various tool categories and interfaces you can use.

Table 1: Tool categories and interfaces

Category	Names
Linux tools	bash, zsh, sh, tmux, and dig
Azure tools	Azure CLI and Azure classic CLI, AzCopy, Azure Functions CLI, Service Fabric CLI, Batch Shipyard, and blobxfer
Text editors	code (Cloud Shell editor), vim, nano, and emacs
Source control	git
Build tools	make, maven, npm, and pip
Containers	Docker Machine, Kubectl, Helm, and DC/OS CLI
Databases	MySQL client, PostgreSql client, sqlcmd Utility, and mssql-scripter
Other	iPython Client, Cloud Foundry CLI, Terraform, Ansible, Chef InSpec, Puppet Bolt, HashiCorp Packer, and Office 365 CLI

Demonstration - Use Cloud Shell

In this demonstration, you'll learn how to experiment with Azure Cloud Shell.

Configure the Cloud Shell

1. Access the **Azure Portal**.
2. Select the **Cloud Shell** icon on the banner.
3. On the **Welcome to Azure Cloud Shell** page, notice your selections for Bash or PowerShell. Select **PowerShell**.

4. The Azure Cloud Shell requires an Azure file share to persist files. If you have time, select **Learn more** to obtain information about the Cloud Shell storage and the associated pricing.
5. Select your **Subscription**, and then select **Create Storage**.

Experiment with Azure PowerShell

1. Wait for your storage to be created and your account to be initialized.
2. At the PowerShell prompt, enter **Get-AzSubscription** to review your subscriptions.
3. Enter **Get-AzResourceGroup** to review resource group information.

Experiment with the Bash shell

1. Use the drop-down list to switch to the **Bash** shell and confirm your choice.
2. At the Bash shell prompt, enter **az account list** to review your subscriptions. Also, try tab completion.
3. Enter **az resource list** to review resource information.

Experiment with the Cloud Editor

1. To use the Cloud Editor, enter **code .**. You can also select the curly braces icon.
2. Select a file from the **navigation** pane; for example, **.profile**.
3. On the editor banner, notice the selections for **Settings**, such as **Text Size**, **Font**, and **Upload/ Download files**.
4. Notice the ellipses (**...**) for **Save**, **Close Editor**, and **Open File**.
5. After experimenting, you can close the Cloud Editor.
6. Close Azure Cloud Shell.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

If you want to use the Azure Cloud Shell to manage Azure resources, what should you install on your computer?

Question 2

To be able to run Azure Cloud Shell, what Azure resources are required?

Manage Azure VMs with PowerShell

Lesson overview

Azure virtual machines (VMs) provide a fully configurable and flexible computing environment. You can create and manage these VMs by using the Azure portal, the Windows PowerShell with Az module, or the Cloud Shell environment. In this lesson, you'll learn how to create and manage Azure VMs by using PowerShell.

Lesson objectives

After completing this lesson, you'll be able to:

- Create Azure VMs with Windows PowerShell.
- Manage Azure VMs and their resources with Windows PowerShell.

Creating Azure VMs with Windows PowerShell

To create a new Azure virtual machine (VM) with PowerShell commands, you can use the locally installed Windows PowerShell with Az module, or you can use the Cloud Shell environment that's available in Azure portal. If you choose to use your locally installed PowerShell, it's recommended that you use Windows PowerShell 7.1. You should also install the Az module, so you can have Azure-related commands available. Also, when using locally installed PowerShell, you first need to use the **Connect-AzAccount** command to authenticate and connect to your Azure tenant. When you run this command in your PowerShell environment, you'll be prompted to authenticate. You need to use credentials from your Azure tenant, with privileges that allow you to create the resources needed for Azure VMs.

To create an Azure VM, you need to perform the following tasks:

1. Create a resource group.
2. Create an Azure VM.
3. Connect to the Azure VM.

Create a resource group

An Azure resource group is a logical container into which Azure resources are deployed and managed. You must create a resource group before you create a VM. In the following example, a resource group named **myResourceGroup** is created in the West Europe region:

```
New-AzResourceGroup -ResourceGroupName "myResourceGroup" -Location "west-
europe"
```

The resource group is later used when creating or modifying a VM or the resources attached to a VM.

Create an Azure VM

The **New-AzVM** cmdlet creates a VM in Azure. This cmdlet uses a VM object as input. Use the **New-AzVMConfig** cmdlet to create a virtual machine object. When creating a VM, several options are available, such as operating system image, network configuration, and administrative credentials. You can use other cmdlets to configure the VM, such as **Set-AzVMOperatingSystem**, **Set-AzVMSourceImage**, **Add-AzVMNetworkInterface**, and **Set-AzVMOSDisk**.

Additional reading: For more information about the parameters that you can use with the **New-AzVM** command, refer to **New-AzVM**[7].

Before you run the **New-AzVM** command, you need to specify the credentials that you'll use to sign in to the newly created Azure VM. The credentials that you specify during this process will be assigned with local administrative privileges on the VM you're creating. It's easiest to store these credentials in a variable, before creating a new Azure VM. To do this, run this command:

```
$cred = Get-Credential
```

When you run this command, you'll be prompted to provide the username and password for the Azure VM. These credentials will be stored in the `$cred` variable.

After you store administrative credentials, you need to define parameters for the new VM. You don't need to provide all the parameters that **New-AzVM** supports. Most of them are optional, and if you don't provide them, their default values will be selected. You can also change most of these parameters later.

You can choose to provide VM parameters directly with the **New-AzVM** command, or you can define these parameters in a variable, and then use this variable with the **New-AzVM** command.

The following code depicts an example of defining VM parameters:

```
$vmParams = @{
  ResourceGroupName = 'myResourceGroup'
  Name = 'TestVM'
  Location = 'westeurope'
  ImageName = 'Win2016Datacenter'
  PublicIpAddressName = 'TestPublicIp'
  Credential = $cred
  OpenPorts = 3389
}
```

When you define VM parameters as the previous example depicts, you can then use the following command to create a new Azure VM, based on these parameters:

```
New-AzVM @vmParams
```

Alternatively, you can also choose to provide VM parameters directly with **New-AzVM** as in the following example:

```
New-AzVm `
    -ResourceGroupName "myResourceGroup"
    -Name "myVM"
    -Location "EastUS"
    -VirtualNetworkName "myVnet"
    -SubnetName "mySubnet"
    -SecurityGroupName "myNetworkSecurityGroup"
    -PublicIpAddressName "myPublicIpAddress"
    -Credential $cred
```

7 https://aka.ms/new-azvm

Connect to the Azure VM

After a new Azure VM is created, you need to connect to it to verify the deployment. After the deployment has completed, create a remote desktop connection with the VM.

Run the following commands to return the public IP address of the VM. Take note of this IP address so you can connect to it with your browser to test web connectivity in a future step.

```
Get-AzPublicIpAddress -ResourceGroupName "myResourceGroup"  | Select IpAd-
dress
```

To create a remote desktop session with the VM, use the following command on your local machine. Replace the IP address with the *publicIPAddress* of your VM. When prompted, enter the credentials you used when creating the VM.

```
mstsc /v:<publicIpAddress>
```

When you run this command, you'll be prompted for credentials to connect to the VM. In the Windows Security window, select **More choices**, and then select **Use a different account**. Enter the username and password you created for the VM, and then select **OK**. After you connect to your Azure VM through Remote Desktop Protocol (RDP), you'll be able to manage it the same way as any other computer.

Managing Azure VMs with Windows PowerShell

Besides using Windows PowerShell to create new Azure VMs, you can also use PowerShell commands to manage, modify, and remove Azure VMs and the resources related to Azure VMs. This topic covers some of the most common tasks for managing and modifying Azure VMs with PowerShell.

Modifying VM sizes

The VM size determines the amount of compute resources such as CPU, GPU, and memory that are made available to the VM. You should create VMs using a VM size that's appropriate for the workload. If a workload increases, you can also resize existing VMs.

To review a list of VM sizes available in a particular region, use the **Get-AzVMSize** command. For example:

```
Get-AzVMSize -Location "EastUS"
```

After a VM has been deployed, you can resize it to increase or decrease resource allocation. Before resizing a VM, check if the size you want is available on the current VM cluster. The **Get-AzVMSize** command returns a list of sizes:

```
Get-AzVMSize -ResourceGroupName "myResourceGroup" -VMName "myVM"
```

If your preferred size is available, you can resize the VM from a powered-on state; however, it's rebooted during the operation. The following example depicts how to change VM size to the **Standard_DS3_v2** size profile:

```
$vm = Get-AzVM -ResourceGroupName "myResourceGroupVM" -VMName "myVM"
$vm.HardwareProfile.VmSize = "Standard_DS3_v2"
Update-AzVM -VM $vm -ResourceGroupName "myResourceGroup"
```

Management tasks

During the lifecycle of a VM, you might want to run management tasks such as starting, stopping, or deleting a VM. Additionally, you might want to create scripts to automate repetitive or complex tasks. You can use Azure PowerShell to perform many common management tasks by using the command line or scripts.

To stop and deallocate a VM with **Stop-AzVM**, you can run the following command:

```
Stop-AzVM -ResourceGroupName "myResourceGroup" -Name "myVM" -Force
```

To start a VM, you can run the following command:

```
Start-AzVM -ResourceGroupName "myResourceGroup" -Name "myVM"
```

If you want to delete everything inside of a resource group, including VMs, you can run the following command:

```
Remove-AzResourceGroup -Name "myResourceGroupVM" -Force
```

Adding disks to Azure VMs

When you create an Azure VM, two disks are automatically attached to the VM:

- Operating system disk. These disks can be sized up to 4 terabytes and host the VM's operating system.
- Temporary disk. These disks use a solid-state drive that's located on the same Azure host as the VM. Temporary disks are highly performant and might be used for operations such as temporary data processing.

You can add additional data disks for installing applications and storing data. You should use data disks in any situation that requires durable and responsive data storage. The size of the VM determines how many data disks can be attached to it.

To add a data disk to an Azure VM after you create it, you need to define disk configuration by using the **New-AzDiskConfig** command. You then need to use the **New-AzDisk** and **Add-AzVMDataDisk** commands to add a new disk to the VM, as the following example depicts:

```
$diskConfig = New-AzDiskConfig -Location "EastUS" -CreateOption Empty -Disk-
SizeGB 128
$dataDisk = New-AzDisk -ResourceGroupName "myResourceGroupDisk" -DiskName
"myDataDisk" -Disk $diskConfig

$vm = Get-AzVM -ResourceGroupName "myResourceGroupDisk" -Name "myVM"
$vm = Add-AzVMDataDisk -VM $vm -Name "myDataDisk" -CreateOption Attach
-ManagedDiskId $dataDisk.Id -Lun 1

Update-AzVM -ResourceGroupName "myResourceGroupDisk" -VM $vm
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

What PowerShell command should you use to create a new resource group in Azure?

Question 2

What resource do you need to connect to an Azure virtual machine (VM)?

Manage storage and subscriptions

Lesson overview

You can use the PowerShell Az module to manage various Azure resources. One of the most common usages, besides managing Azure VMs, is to manage Azure storage accounts and Azure subscriptions. In this lesson, you'll learn about managing Azure storage accounts and Azure subscriptions with PowerShell.

Lesson objectives

After completing this lesson, you'll be able to:

- Manage Azure storage accounts with PowerShell.
- Manage Azure subscriptions with PowerShell.

Managing storage with Azure PowerShell

You can use Azure PowerShell to manage Azure-related storage. Before you start managing your storage, you should first create a storage account, if you don't have one. Usually, storage accounts are created automatically when you create other Azure resources such as Azure virtual machines (VMs).

You can create a standard, general-purpose storage account with locally redundant storage (LRS) replication by using **New-AzStorageAccount**. Next, get the storage account context that defines the storage account you want to use. When acting on a storage account, reference the context, instead of repeatedly passing in the credentials. Use the following example to create a storage account called **mystorageaccount** with LRS and blob encryption, which is enabled by default.

```
$storageAccount = New-AzStorageAccount -ResourceGroupName $resourceGroup `
  -Name "mystorageaccount" `
  -SkuName Standard_LRS `
  -Location $location `

$ctx = $storageAccount.Context
```

Blobs are always uploaded into a container. You can organize groups of blobs the way you organize your files on your computer in folders.

Set the container name, and then create the container by using **New-AzStorageContainer**. Set the blob permissions to allow public access of the files. The container name in the following example is **quickstartblobs**.

```
$containerName = "quickstartblobs"
New-AzStorageContainer -Name $containerName -Context $ctx -Permission blob
```

You can use the **Set-AzStorageAccount** cmdlet to modify an Azure Storage account. You can use this cmdlet to modify the account type, update a customer domain, or set tags on a Storage account.

For example, to set the storage account type you should use the following command:

```
Set-AzStorageAccount -ResourceGroupName "MyResourceGroup" -AccountName
"mystorageaccount" -Type "Standard_RAGRS"
```

To set custom domain for existing storage account, you can use the following command:

```
Set-AzStorageAccount -ResourceGroupName "MyResourceGroup" -AccountName
"mystorageaccount" -CustomDomainName "www.contoso.com" -UseSubDomain $True
```

Additional reading: To learn more about the available cmdlets for managing Azure storage, refer to **Az. Storage**[8].

Managing Azure subscriptions with Azure PowerShell

Most Azure users will only ever have a single subscription. However, if you're part of more than one organization or your organization has divided up access to certain resources across groupings, you might have multiple subscriptions within Azure.

In Azure PowerShell, accessing the resources for a subscription requires changing the subscription associated with your current Azure session. You can do this by modifying the active session context, which is the information about which tenant, subscription, and user the cmdlets should be run against. To change subscriptions, you need to first retrieve an Azure PowerShell Context object with **Get-AzSubscription**, and then change the current context with **Set-AzContext**.

The **Get-AzSubscription** cmdlet gets the subscription ID, subscription name, and home tenant for subscriptions that the current account can access.

To get all Azure subscriptions active on all tenants, run the following command:

```
Get-AzSubscription

Name                               Id                      TenantId
State
----                               --                      --------
-----
Subscription1                      yyyy-yyyy-yyyy-yyyy     aaaa-aaaa-aaaa-
aaaa             Enabled
Subscription2                      xxxx-xxxx-xxxx-xxxx     aaaa-aaaa-aaaa-
aaaa             Enabled
Subscription3                      zzzz-zzzz-zzzz-zzzz     bbbb-bbbb-bbbb-
bbbb             Enabled
```

To focus on subscriptions assigned to a specific tenant, run the following command:

```
Get-AzSubscription -TenantId "aaaa-aaaa-aaaa-aaaa"

Name                               Id                      TenantId
State
----                               --                      --------
-----
Subscription1                      yyyy-yyyy-yyyy-yyyy     aaaa-aaaa-aaaa-
aaaa             Enabled
Subscription2                      xxxx-xxxx-xxxx-xxxx     aaaa-aaaa-aaaa-
aaaa             Enabled
```

8 https://aka.ms/az-storage-2

The **Set-AzContext** cmdlet sets authentication information for cmdlets that you run in the current session. The context includes tenant, subscription, and environment information.

To set the subscription context, run the following command:

```
Set-AzContext -Subscription "xxxx-xxxx-xxxx-xxxx"

Name    Account             SubscriptionName    Environment         Tenan-
tId
----    -------             ----------------    -----------         -------
-
Work    test@outlook.com    Subscription1       AzureCloud
xxxxxxxx-x...
```

The next example depicts how to get a subscription in the currently active tenant and set it as the active session:

```
$context = Get-AzSubscription -SubscriptionId ...
Set-AzContext $context
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

What do you need to have in place before configuring storage in Azure?

Question 2

Which command should you use to get information about the Azure subscriptions that are associated with your account?

Module 09 lab and review

Lab: Azure resource management with PowerShell

Scenario

You're a system administrator for the London branch office of Adatum Corporation. You need to evaluate Azure platform to run virtual machines (VMs) and other resources for your company. As a part of your evaluation, you also want to test PowerShell administration of Azure-based resources.

Objectives

After completing this lab, you'll be able to:

- Install Az module for Windows PowerShell.
- Run and use Azure Cloud Shell environment.
- Manage Azure VMs and disks by using PowerShell.

Estimated time: 60 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

What should you use to migrate scripts from AzureRM to the Az PowerShell module?

Question 2

If you want to manage Azure Active Directory (Azure AD) through PowerShell, what module (or modules) should you install?

Question 3

What are the environments you can use in Azure Cloud Shell?

Question 4

If you want to define disk configuration for a new disk on an Azure virtual machine (VM), which command should you use?

Answers

Question 1

Besides using Azure PowerShell in interactive mode, what's the other mode you can use it in to run commands?

You can also use Azure PowerShell in the scripting mode, where you can run a script that consists of multiple commands.

Question 2

Is it possible to use the Az module for PowerShell on Mac computers?

Yes, you can use the Az module for PowerShell on MacOS.

Question 3

What's the preferred way to install the Az module for PowerShell? What version of PowerShell is recommended for this module?

It's recommended that you use the `Install-Module` command on Windows PowerShell 7.1 to install Az module.

Question 1

If you want to use the Azure Cloud Shell to manage Azure resources, what should you install on your computer?

You don't have to install anything. Azure Cloud Shell is an interactive, browser-accessible shell for managing Azure resources.

Question 2

To be able to run Azure Cloud Shell, what Azure resources are required?

Azure Cloud Shell requires a resource group, storage account, and Azure file share.

Question 1

What PowerShell command should you use to create a new resource group in Azure?

To create a new resource group in Azure, you should use the `New-AzResourceGroup` command.

Question 2

What resource do you need to connect to an Azure virtual machine (VM)?

To connect to an Azure VM, you need the public IP address or public DNS name of the Azure VM.

Question 1

What do you need to have in place before configuring storage in Azure?

Before configuring storage in Azure, you need to create a storage account.

Question 2

Which command should you use to get information about the Azure subscriptions that are associated with your account?

To get information about the Azure subscriptions that are associated with your account, you should use the `Get-AzSubscription` command.

Question 1

What should you use to migrate scripts from AzureRM to the Az PowerShell module?

The recommended option to migrate from AzureRM to the Az PowerShell module is automatic migration. For this, you need to install the AzureRM to Az migration toolkit.

Question 2

If you want to manage Azure Active Directory (Azure AD) through PowerShell, what module (or modules) should you install?

You can use the Azure Active Directory Module for Windows PowerShell or the Azure Active Directory PowerShell for Graph module to manage Azure AD through PowerShell.

Question 3

What are the environments you can use in Azure Cloud Shell?

You can use Azure Cloud Shell in the PowerShell or Bash environment.

Question 4

If you want to define disk configuration for a new disk on an Azure virtual machine (VM), which command should you use?

To define disk configuration for a new disk on an Azure VM, you should use the `New-AzDiskConfig` command.

Module 10 Managing Microsoft 365 services with PowerShell

Manage Microsoft 365 user accounts, licenses, and groups with PowerShell

Lesson overview

Microsoft 365 includes multiple cloud services, but the core of Microsoft 365 is Azure AD. Azure AD provides identity management for all the services in Microsoft 365. To give access to the services in Microsoft 365, you need to create user accounts and then assign licenses that provide access to the services. Within Microsoft 365, you can use roles to give users permissions to manage Microsoft 365 services.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the benefits of using PowerShell for Microsoft 365.
- Explain how to connect to a Microsoft 365 tenant with PowerShell.
- Explain how to create and manage users in Microsoft 365 with PowerShell.
- Explain how to create and manage groups in Microsoft 365 with PowerShell.
- Explain how to manage roles in Microsoft 365 with PowerShell.
- Explain how to manage licenses in Microsoft 365 with PowerShell.

Benefits of using PowerShell for Microsoft 365

The Microsoft 365 admin center is a web-based console that you can use to manage users, groups, licensing, and other tenant-level configuration settings. When you manage users and groups from here, the properties you modify might be part of Azure AD, Exchange Online, or another service. This console provides a unified interface for some common management tasks.

Links to service-specific, web-based consoles enable you to perform more detailed configuration of Microsoft 365 services. You can use these links to manage services such as Azure AD, Exchange Online, Microsoft Teams, and SharePoint Online. With the service-specific consoles, you can perform tasks such as:

- Creating users and group.
- Modifying email addresses.
- Setting organizational defaults for Teams.
- Configuring external sharing settings for SharePoint Online.

Note: You can access the Microsoft 365 admin console at `https://admin.microsoft.com`.

The service-specific, web-based consoles are intuitive and easy to use, but they don't provide access to all possible configuration options. There are many useful configuration options that you can review and configure only by using PowerShell cmdlets. For example, in Exchange Online you can use PowerShell to review the permissions assigned to and configured on a calendar in a mailbox. In the web-based console, you can only review mailbox-level permissions.

Using PowerShell to manage Microsoft 365 services also provides many of the same benefits as using PowerShell to manage local resources. You can:

- Use a query for objects matching certain criteria, and then generate reports.
- Use the pipeline to perform complex operations.
- Automate bulk processes.
- Manage multiple services simultaneously.

Connecting to the Microsoft 365 tenant with PowerShell

You can manage Microsoft 365 by using the Azure AD PowerShell for Graph (AzureAD) module or the Microsoft Azure Active Directory Module for Windows PowerShell (MSOnline) module. The Azure AD PowerShell for Graph module is the newer module and is generally preferred over the Azure Active Directory Module for Windows PowerShell module. However, some functionality in the Azure Active Directory Module for Windows PowerShell module is not replicated in the Azure AD PowerShell for Graph module. Depending on your task, you might need to install and use both modules.

Azure AD PowerShell for Graph

You can use the Azure AD PowerShell for Graph module in Windows PowerShell 5.1 or newer, including PowerShell 6 and PowerShell 7. Azure AD PowerShell for Graph is the newest PowerShell module for managing Microsoft 365 and has some features that aren't available in the older Azure Active Directory Module for Windows PowerShell module. As a result, when using PowerShell to manage Microsoft 365, the Azure AD module is preferred.

Note: All cmdlets provided by the Azure AD PowerShell for Graph module have **AzureAD** in the name of the cmdlet. For example, **Get-AzureADUser**.

The AzureAD cmdlets use the Azure AD Graph API to access and modify data. The Azure AD Graph API is a REST API that can be accessed directly by web requests. The AzureAD cmdlets simplify this process for you.

You can install the Azure AD PowerShell for Graph module from the PowerShell Gallery by running the following command:

```
Install-Module AzureAD
```

After the Azure AD PowerShell for Graph module is installed, you can connect to Microsoft 365 by running the following command:

```
Connect-AzureAD
```

When you connect to Microsoft 365, you're prompted for a username and password to sign in. You need to sign in with a user account that has sufficient privileges to perform the actions. You might also be prompted for multifactor authentication.

Note: For tenants in China or Germany, you need to use the *-AzureEnvironmentName* parameter and specify **AzureChinaCloud** or **AzureGermanyCloud** respectively. For secure US government tenants, you need to specify **AzureUSGovernment**.

Azure AD Module for Windows PowerShell

The Azure Active Directory Module for Windows PowerShell module can be used in Windows PowerShell 5.1 or PowerShell 6. This is the original module for managing Microsoft 365. Some tasks that you can perform by using the Azure Active Directory Module for Windows PowerShell module were never added to the Azure AD PowerShell for Graph module. There are also some tasks that are easier to perform by using cmdlets from the Azure Active Directory Module for Windows PowerShell module. Microsoft isn't developing new features for this module, but you can continue to use these cmdlets.

Note: All of the cmdlets provided by the Azure Active Directory Module for Windows PowerShell module have **Msol** in the name of the cmdlet. For example, **Get-MsolUser**.

You can install the Azure Active Directory Module for Windows PowerShell module from the PowerShell Gallery by running the following command:

```
Install-Module MSOnline
```

After the Azure Active Directory Module for Windows PowerShell module is installed, you connect to Microsoft 365 by running the following command:

```
Connect-MSOnline
```

Microsoft Graph PowerShell SDK

Microsoft has announced that future management interface development will be focused on the Microsoft Graph API. This is a web-based API that's separate from the Azure AD Graph API used by the Azure AD PowerShell for Graph module. The Azure AD Graph API will cease updates after June 2022. All future development will be in the Microsoft Graph API.

To access the Microsoft Graph API, you can use **Invoke-WebRequest**, but this process is difficult. To simplify the process of using Microsoft Graph API, you can use the Microsoft Graph PowerShell SDK (Microsoft.Graph).

You can install the Microsoft.Graph module from the PowerShell Gallery by running the following command:

```
Install-Module Microsoft.Graph
```

After the Microsoft.Graph module is installed, you can connect to Microsoft 365 by running the following command:

```
Connect-MgGraph -Scopes "User.ReadWrite.All"
```

When you connect by using Microsoft.Graph, the scope specifies the permissions required. If your user account hasn't been assigned the necessary permissions already, an administrator needs to grant the permissions. There are many permission scopes available based on the type of object to be managed and the actions allowed.

The Microsoft.Graph module creates cmdlets for the available Microsoft Graph options. This avoids much of the complexity that's typically required when using web-based APIs where you need to understand which URL to use and send data in a specific format. The Microsoft.Graph module provides you with cmdlets and parameters as the user interface.

Note: Cmdlets provided by the Microsoft.Graph module have **Mg** in the name of the cmdlet. For example, **Get-MgUser**.

Additional reading: The remainder of this module focuses on using the AzureAD and Azure Active Directory Module for Windows PowerShell modules. For more information about using the Microsoft. Graph module, refer to **Get started with the Microsoft Graph PowerShell SDK**[1].

Azure Cloud Shell

As an alternative to installing and maintaining the AzureAD and Azure Active Directory Module for Windows PowerShell modules in multiple locations, you can use Azure Cloud Shell. Cloud Shell is a prompt with PowerShell functionality that you can access through a web browser. The Microsoft 365 admin center provides a link to open Cloud Shell.

Many PowerShell modules that are used to manage Microsoft 365 services are automatically installed in the shell. You must have an Azure subscription to use Cloud Shell.

Additional reading: For more information about Cloud Shell, refer to **Overview of Azure Cloud Shell**[2].

Managing users in Microsoft 365 with PowerShell

Before users can sign in and begin using Microsoft 365 services, you need to create user accounts for them. After you create the accounts, you might need to modify them. You can manage users in Microsoft 365 by using both Msol and AzureAD cmdlets. In both cases, you need to connect to Microsoft 365 before you can create and manage user accounts.

The following table lists the user attributes that you need to consider when creating user accounts.

Table 1: User attributes

1 https://aka.ms/get-started-with-the-Microsoft-Graph-PowerShell-SDK
2 https://aka.ms/overview-of-azure-cloud-shell-2

Property	Required	Description
DisplayName	Yes	This is the name that displays for users in the web-based management tools.
UserPrincipalName	Yes	This is the name that people use to sign in to Microsoft 365. This is also a unique identifier that you use when performing management tasks with PowerShell cmdlets.
GivenName/FirstName	No	This property can be used by various Microsoft 365 services such as the Exchange Online address book.
SurName/LastName	No	This property can be used by various Microsoft 365 services such as the Exchange Online address book.
Password	No	A password is required to enable a user account. When you create a user with the **New-AzureADUser** cmdlet, you must set a password.
LicenseAssignment	No	This property specifies the licensing plan for the user, which in turn determines which Microsoft 365 services the user can access. You can assign licenses after you create the user.
UsageLocation	No	The usage location is a two-character country code. You can't assign a license if you haven't set a usage location.

Note: For more information about licensing user accounts, refer to Lesson 1 Unit 7, **Managing licenses in Microsoft 365 with PowerShell**.

Manage users with AzureAD cmdlets

You can create user accounts in Microsoft 365 by using the **New-AzureADUser** cmdlet. The following code block depicts how you can use this cmdlet to create a new user account and set a password. The password is stored in an object required for that purpose. In this example, the *-AccountEnabled* parameter is set to `$true` to enable the account and allow the user to sign in. The *-PasswordProfile* and *-AccountEnabled* parameters are required:

```
$UserPassword=New-Object -TypeName Microsoft.Open.AzureAD.Model.Password-
Profile
$UserPassword.Password="Pa55w.rd"
New-AzureADUser -DisplayName "Abbie Parsons" -GivenName "Abbie" -SurName
"Parsons" -UserPrincipalName AbbieP@adatum.com -UsageLocation US -Password-
```

```
Profile $UserPassword -AccountEnabled $true
```

You can query a list of user accounts in Microsoft 365 by using the **Get-AzureADUser** cmdlet. The following table lists commonly used parameters for this cmdlet.

Table 2: Parameters for the Get-AzureADUser cmdlet

Parameter	Description
-ObjectID	Specifies the user principle name (UPN) or ObjectID of a specific user account to retrieve. Both properties are unique identifiers for a user account in Microsoft 365.
-Filter	Specifies a filter in oPath format that you can use to query a specific set of user accounts.
-SearchString	Specifies a string that is matched against the start of the **DisplayName** and **UserPrincipalName** attributes.
-All	By default, **Get-AzureADUser** returns only 100 results. If you set the -All parameter to `$true` then all results are returned. The default value for this parameter is `$false`.
-Top	When the -All parameter is `$false`, you can use -Top to specify the maximum number of results to return.

Note: The oPath format used for filters doesn't support using wildcards. If you need to perform a wildcard search of user accounts, you need to retrieve all the user accounts and then filter them by using the **Where-Object** cmdlet.

The following example depicts how to query a single Microsoft 365 user account:

```
Get-AzureADUser -ObjectId AbbieP@adatum.com
```

The following example depicts how to query all the user accounts in a Microsoft 365 tenant:

```
Get-AzureADUser -All $true
```

The following table lists other commonly used cmdlets for user account management.

Table 3: AzureAD cmdlets for user account management

Cmdlet	Description
Set-AzureADUser	Modifies the properties of a user account.
Remove-AzureADUser	Deletes a user account.
Set-AzureADUserPassword	Sets the password for a user account.
Get-AzureADMSDeletedDirectoryObject	Lists soft-deleted user accounts.

Manage users with Msol cmdlets

You can create new user accounts in Microsoft 365 by using the **New-MsolUser** cmdlet. When you create an account with **New-MsolUser**, it's automatically enabled. The password is supplied as a string. If you don't specify a password, a randomly generated password is set automatically. The following code block depicts how you can create a new user account and set a password by using the cmdlet:

```
New-MsolUser -DisplayName "Abbie Parsons" -FirstName "Abbie" -LastName
"Parsons" -UserPrincipalName AbbieP@adatum.com -Password "Pa55w.rd"
```

You can query a list of user accounts in Microsoft 365 by using the **Get-MsolUser** cmdlet. The following table lists commonly-used parameters for this cmdlet.

Table 4: Parameters for the Get-MsolUser cmdlet

Parameter	Descriptions
-ObjectID	Specifies the ObjectID for a specific user account to retrieve.
-UserPrincipalName	Specifies the UPN for a specific user account to retrieve.
-SearchString	Defines a string that the display name and email addresses are searched for. This behaves as a wildcard search for the specified string that can match any part of the display name or email addresses.
-All	Retrieves all available results instead of the default 500 results.
-MaxResults	Specifies to return more than the default 500 results when the -All parameter is not used.
-ReturnDeletedUsers	Returns only soft-deleted users.

The **Get-MsolUser** cmdlet doesn't have a generic filtering parameter. Instead, there are parameters for filtering based on specific attributes. For example, you can use the *-City* parameter to filter based on the **City** attribute configured in user accounts. To filter based on attributes that don't have a corresponding parameter, you need to retrieve all user accounts and then use the **Where-Object** cmdlet.

The following example depicts how to query a single Microsoft 365 user account:

```
Get-MsolUser -UserPrincipalName AbbieP@adatum.com
```

The following example depicts how to query all user accounts in a Microsoft 365 tenant:

```
Get-MsolUser -All
```

The following table lists other commonly used cmdlets for user account management.

Table 5: Msol cmdlets for user account management

Cmdlet	Description
Set-MsolUser	Modifies user account properties.
Remove-MsolUser	Deletes a user account.
Set-MsolUserPassword	Sets a user account's password.
Set-MsolUserPrincipalName	Changes the UPN for a user account.
Restore-MsolUser	Restores a soft-deleted user account.

Synchronized users

Users that you create in Microsoft 365 with Windows PowerShell are cloud users. Many organizations use Azure AD Connect to synchronize users and groups from on-premises AD DS to Microsoft 365. These users and groups are created by Azure AD Connect. and as such you can't delete them directly in Microsoft 365. Instead, you need to delete the object in AD DS, and the deletion is synchronized to Microsoft 365.

When objects are synchronized from AD DS to Microsoft 365, the value of some attributes in AD DS is authoritative. This means you can't modify the attribute's value in Microsoft 365. Instead, you need to modify the value in AD DS, and then the modified value synchronizes to Microsoft 365. Attempting to modify these attributes in Microsoft 365 will generate an error.

The following list are some of the common attributes for which AD DS is authoritative:

- **UserPrincipalName**
- **DisplayName**
- **AccountEnabled**
- **ProxyAddresses** (email addresses)

Managing groups in Microsoft 365 with PowerShell

Microsoft 365 has multiple types of groups that you can use to provide access to resources or send email. Each group type serves a different purpose, and you should select the correct type for your needs. The group types available in Microsoft 365 are listed in the following table.

Table 1: Types of groups in Microsoft 365

Group type	Description
Microsoft 365 group	Microsoft 365 groups are used for collaboration between users, both inside and outside your company. With each Microsoft 365 group, members get a group email and shared workspace for conversations, files, and calendar events, Microsoft Stream, and Microsoft Planner.
Distribution group	Distribution groups are used for sending notifications to a group of people. They can receive external email if enabled by the administrator.
Security group	Security groups are used for granting access to Microsoft 365 resources, such as SharePoint. They can make administration easier because you only need to administer the group rather than adding users to each resource individually.
Mail-enabled security group	A mail-enabled security group has the same functionality as a security group for assigning permissions, but it can also be used to send email messages just as you can with a distribution group.

Additional reading: For a detailed comparison of group types, refer to **Compare groups**[3].

Note: There are no AzureAD or Msol cmdlets for managing Microsoft 365 groups. The cmdlets for managing Microsoft 365 groups are part of Exchange Online, because these groups include a mailbox. In the Exchange Online cmdlets, Microsoft 365 groups are referred to as *unified groups*.

Managing groups with AzureAD cmdlets

You can create distribution groups, security groups, and mail-enabled security groups by using the **New-AzureADGroup** cmdlet. The type of group created depends on how you use the *-MailEnabled* and *-SecurityEnabled* parameters. The following example creates a mail-enabled security group:

```
New-AzureADGroup -DisplayName "Marketing Group" -MailEnabled $true -Securi-
tyEnabled $true -MailNickname MarketingGrp
```

When you create a new group, the group is assigned an ObjectID. The ObjectID is a unique identifier for the group. You need to use the ObjectID for the group with management cmdlets. You can't use a display name to refer to a group because it isn't guaranteed to be unique. Use the **Get-AzureADGroup** cmdlet to identify the ObjectID for a group that you want to manage.

Other AzureAD cmdlets that you can use to manage groups are listed in the following table.

Table 2: AzureAD cmdlets for group management

Cmdlet	Description
Get-AzureADGroup	Queries for distribution groups, security groups, and mail-enabled security groups. This cmdlet supports using a filter or search string.
Set-AzureADGroup	Modifies the properties of distribution groups, security groups, and mail-enabled security groups.
Remove-AzureADGroup	Deletes distribution groups, security groups, and mail-enabled security groups.
Get-AzureADGroupMember	Queries the membership of distribution groups, security groups, and mail-enabled security groups.
Add-AzureADGroupMember	Adds a member to distribution groups, security groups, and mail-enabled security groups.
Remove-AzureADGroupMember	Removes a member from distribution groups, security groups, and mail-enabled security groups.
Get-AzureADGroupOwner	Queries the owners of distribution groups, security groups, and mail-enabled security groups.
Add-AzureADGroupOwner	Adds an owner to distribution groups, security groups, and mail-enabled security groups.
Remove-AzureADGroupOwner	Removes an owner from distribution groups, security groups, and mail-enabled security groups.

Managing groups with Msol cmdlets

You can use the **New-MsolGroup** cmdlet to create security groups. There are no Msol cmdlets to create distribution groups or mail-enabled security groups. The following example create a new security group:

3 https://aka.ms/compare-groups

```
New-MsolGroup -DisplayName "Marketing Group"
```

Other Msol cmdlets that you can use to manage groups are listed in the following table.

Table 3: Msol cmdlets for managing groups

Cmdlet	Description
Get-MsolGroup	Queries for distribution groups, security groups, and mail-enabled security groups. You can filter results based on group type or a search string for the display name.
Set-MsolGroup	Modifies the properties of distribution groups, security groups, and mail-enabled security groups.
Remove-MsolGroup	Deletes distribution groups, security groups, and mail-enabled security groups.
Get-MsolGroupMember	Queries the membership of distribution groups, security groups, and mail-enabled security groups.
Add-MsolGroupMember	Adds a member to distribution groups, security groups, and mail-enabled security groups.
Remove-MsolGroupMember	Removes a member from distribution groups, security groups, and mail-enabled security groups.

Synchronized groups

Groups that you create in Microsoft 365 with Windows PowerShell are cloud groups. Many organizations use Azure AD Connect to synchronize users and groups from on-premises AD DS to Microsoft 365. These users and groups are created by Azure AD Connect, and you can't them delete directly in Microsoft 365. Instead, you need to delete the object in AD DS, and the deletion is synchronized to Microsoft 365.

When objects are synchronized from AD DS to Microsoft 365, the value of some attributes in AD DS is authoritative. This means you can't modify the attribute value in Microsoft 365. Instead, you need to modify the value in AD DS, and the modified value synchronizes to Microsoft 365. If you attempt to modify these attributes in Microsoft 365, an error is generated.

Group membership from on-premises AD DS is authoritative. You can't edit the membership of a synchronized group. Instead, you need to modify the membership in the on-premises AD DS group.

Managing roles in Microsoft 365 with PowerShell

In Microsoft 365, roles are used to assign administrative permissions to user accounts. The Global Administrator role gives users permission to manage all aspects of Microsoft 365. Other roles allow users to manage only a subset of Microsoft 365 features. The following table lists some of the Microsoft 365 roles that can be assigned to users.

Table 1: Microsoft 365 roles

Role	Description
Global Reader	Can review settings for all aspects of Microsoft 365.

Role	Description
Exchange Administrator	Can review and manage settings for Exchange Online.
Helpdesk admin	Can reset passwords for non-administrative accounts, force non-administrative users to sign out, manage service requests, and monitor service health.
Service support admin	Can open and manage service requests, review and share message center posts, and monitor service health.
SharePoint Administrator	Can review and manage settings for SharePoint Online.
Teams Administrator	Can review and manage settings for Microsoft Teams.
User Administrator	Can review and manage user accounts.

Note: Role names can vary depending on whether you review them in the web-based admin consoles, AzureAD cmdlets, or Msol cmdlets.

Managing roles with AzureAD cmdlets

AzureAD cmdlets require you to identify whether a role is already in use before you can assign it to a user. If no users have been assigned to a role, then it exists only as a template, and you need to enable the role before you can add users to it. You can use the **Get-AzureADDirectoryRole** cmdlet to review the roles that are enabled. Use the **Get-AzureADDirectoryRoleTemplate** cmdlet to review the roles that aren't yet enabled.

The following example depicts how to enable the User Administrator role. When you enable the role, you need to refer to the object ID of the template:

```
$roleTemplate = Get-AzureADDirectoryRoleTemplate | Where {$_.displayName
-eq 'User Administrator'}
Enable-AzureADDirectoryRole -RoleTemplateId $roleTemplate.ObjectId
```

After you enable the role, you can add a role member to assign administrative permissions by using the **Add-AzureADDirectoryRoleMember** cmdlet. The following example depicts how to add an account to the User Administrator role. The *-ObjectId* parameter refers to the ObjectID of the role. The *-RefObjectId* parameter refers to the ObjectID of the user account:

```
$user = Get-AzureADUser -ObjectID AbbieP@adatum.com
$role = Get-AzureADDirectoryRole | Where {$_.displayName -eq 'User Adminis-
trator'}
Add-AzureADDirectoryRoleMember -ObjectId $role.ObjectId -RefObjectId $user.
ObjectID
```

To list existing members of a role, use **Get-AzureADDirectoryRoleMember**. To remove a member from a role, use **Remove-AzureADDirectoryRoleMember**.

Managing roles with Msol cmdlets

The Msol cmdlets don't differentiate between roles that are activated and those that aren't. You can use **Get-MsolRole** to review a list of all roles. The cmdlets that you can use to manage role membership are listed in the following table.

Table 2: Cmdlets to manage role membership

Cmdlet	Description
Add-MsolRoleMember	Adds a user to a role.
Get-MsolRoleMember	Queries members of a role.
Remove-MsolMember	Removes a user from a role.

The following example depicts how to add an account to the User Administrator role. The *-RoleMemberE-mailAddress* parameter can refer to any email address assigned to the user that includes the UPN. You can also use the *-RoleMemberObjectGuid* parameter instead of the *-RoleMemberEmailAddress* parameter:

```
Add-MsolRoleMember -RoleMemberEmailAddress AbbieP@adatum.com -RoleName
'User Administrator'
```

Managing licenses in Microsoft 365 with Power-Shell

After you create user accounts, you need to assign a license to the user account. The license determines which Microsoft 365 services the user has access to. The licenses in the tenant vary depending on which licenses you've purchased.

Some of the common enterprise licenses are:

- Microsoft Office 365 E3
- Office 365 E5
- Microsoft 365 E3
- Microsoft 365 E5

Each license includes multiple services that can be enabled or disabled. For example, an Office 365 E3 license includes access to Exchange Online, Microsoft Teams, SharePoint Online, and others. All services in a license are enabled by default.

When you configure licensing by using Windows PowerShell, you need to refer to the license and service plans by either a string ID or a globally unique identifier (GUID). For example, the Microsoft 365 E3 license has a string ID of SPE_E3 and a GUID of 05e9a617-0261-4cee-bb44-138d3ef5d965. The AzureAD cmdlets for licensing use the GUID, and the Msol cmdlets use the string ID.

Additional reading: It's possible to query the string ID or GUID from your tenant, but you can also find a list of licenses and service plans on **Product names and service plan identifiers for licensing**[4].

Note: Some organizations use group-based licensing, which assigns licenses to user accounts automatically, based on group membership. If a license has been assigned by group-based licensing, you can't modify that license assignment by using PowerShell.

4 https://aka.ms/product-names-and-service-plan-identifiers-for-licensing

Reviewing licenses by using AzureAD cmdlets

You can use the **Get-AzureADSubscribedSku** cmdlet to review the licenses available in your Microsoft 365 tenant. The following example retrieves licenses and displays information about them. The **SkuId** property is the GUID for the license:

```
Get-AzureADSubscribedSku | Select-Object -Property Sku*,ConsumedUnits
-ExpandProperty PrepaidUnits
```

The service plans for a license are stored in the **ServicePlans** property. The following example places all licenses in a variable and then displays the service plans for the first item in the array. The provisioning status for the service plan indicates whether it's enabled or disabled for that user:

```
$sku = Get-AzureADSubscribedSku
$sku[0].ServicePlans
```

Managing licenses by using Azure AD cmdlets

You can use the **Set-AzureADUserLicense** cmdlet to assign a license to a user. Licenses to be added are contained in an **AssignedLicenses** object that you create. For each license that you want to add, you create an **AssignedLicense** object and add it to the **AddLicenses** property of the **AssignedLicenses** object. After the **AssignedLicenses** object is configured, you apply it to the user account. The following example creates a license object for Microsoft 365 E3, and then assigns it to a user:

```
$License = New-Object -TypeName Microsoft.Open.AzureAD.Model.AssignedLi-
cense
$License.SkuId = '05e9a617-0261-4cee-bb44-138d3ef5d965'
$LicensesToAssign = New-Object -TypeName Microsoft.Open.AzureAD.Model.
AssignedLicenses
$LicensesToAssign.AddLicenses = $License
Set-AzureADUserLicense -ObjectId AbbieP@adatum.com -AssignedLicenses $Li-
censesToAssign
```

If you want to disable service plans for a user, you need to add the GUID for the service plans to the **DisabledPlans** property of the license object. The following example depicts how to disable the YAMMER_ENTERPRISE and SWAY service plans in a license:

```
$License.DisabledPlans = '7547a3fe-08ee-4ccb-b430-5077c5041653'
$License.DisabledPlans.Add('a23b959c-7ce8-4e57-9140-b90eb88a9e97')
```

To remove a license, you add a license to the **RemoveLicenses** property of an **AssignedLicenses** object. The following example removes a Microsoft 365 E3 license from a user account:

```
$License = New-Object -TypeName Microsoft.Open.AzureAD.Model.AssignedLi-
cense
$License.SkuId = '05e9a617-0261-4cee-bb44-138d3ef5d965'
$LicensesToAssign = New-Object -TypeName Microsoft.Open.AzureAD.Model.
AssignedLicenses
$LicensesToAssign.RemoveLicenses = $License
Set-AzureADUserLicense -ObjectId AbbieP@adatum.com -AssignedLicenses $Li-
censesToAssign
```

You cannot add multiple licenses to a user that has conflicting components. For example, you can't assign a Microsoft 365 E5 license to a user that already has a Microsoft 365 E3 license. However, you can create an **AssignedLicenses** object that removes the Microsoft 365 E3 license and adds the Microsoft 365 E5 license at the same time.

Reviewing licenses by using Msol cmdlets

You can use the **Get-MsolAccountSku** cmdlet to review the licenses available in your Microsoft 365 tenant. The default output for this cmdlet depicts the **AccountSkuID**, the number of licenses purchased, and the number of licenses assigned. The **AccountSkuId** includes the name of your tenant and is required for assigning or removing licenses.

The service plans for a license are stored in the **ServiceStatus** property. The following example places all licenses in a variable, and then displays the service plans for the first item in the array. The provisioning status for the service plan indicates whether it's enabled or disabled for that user:

```
$sku = Get-MsolAccountSku
$sku[0].ServiceStatus
```

Managing licenses by using Msol cmdlets

You can use the **Set-MsolUserLicense** cmdlet to assign a license to a user. To add the license, you need the **AccountSkuId** for the license you want to assign. The following example assigns a Microsoft 365 E3 to a user:

```
Set-MsolUserLicense -UserPrincipalName "AbbieP@adatum.com" -AddLicenses
"Adatum:SPE_E3"
```

If you want to disable service plans for a user, you need to create a license options object that lists the disabled service plans. The following example depicts how to disable the YAMMER_ENTERPRISE and SWAY service plans for a user:

```
$planList = "YAMMER_ENTERPRISE","SWAY"
$licenseOptions=New-MsolLicenseOptions -AccountSkuId $accountSkuId -Disa-
bledPlans $planList
Set-MsolUserLicense -UserPrincipalName "AbbieP@adatum.com" -LicenseOptions
$licenseOptions
```

Note: You cannot assign a license and license options at the same time. You need to perform the process in two steps, with a delay of a few seconds between the two steps.

To remove a license, you use the *-RemoveLicenses* parameter with the **Set-MsolUserLicense** cmdlet. The following example removes a Microsoft 365 E3 license from a user's account:

```
Set-MsolUserLicense -UserPrincipalName "AbbieP@adatum.com" -RemoveLicenses
"Adatum:SPE_E3"
```

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which cmdlet can you use to connect to Azure AD to create users?

- ☐ `Connect-MicrosoftTeams`
- ☐ `Connect-SPOTenant`
- ☐ `Connect-ExchangeOnline`
- ☐ `Connect-MSOnline`

Question 2

Which role should be assigned to a user to manage all services in Microsoft 365?

- ☐ Global Administrator
- ☐ Global Reader
- ☐ Service Support Admin
- ☐ User Administrator

Manage Exchange Online with PowerShell

Lesson overview

Exchange Online is one of the most commonly used services in Microsoft 365. You can use PowerShell cmdlets to efficiently manage bulk operations and perform tasks that aren't possible through the web-based administrative interface. You can be a better administrator if you're adept at managing Exchange Online with PowerShell. This lesson covers managing mailboxes, resources, and admin roles. However, there are many other aspects you can manage after you've learned these concepts.

Lesson objectives

After completing this lesson, you'll be able to:

- Explain how to connect to Exchange Online by using PowerShell.
- Explain how to manage mailboxes in Exchange Online.
- Explain how to manage resources in Exchange Online.
- Explain how to manage admin roles in Exchange Online.

Connecting to Exchange Online PowerShell

Exchange Online PowerShell is the module that you can use to manage mail-related objects in Exchange Online, such as mailboxes, contacts, and distribution. Some of the information that you can review and manage by using Exchange Online PowerShell, such as email addresses, you can also review in the properties of user objects with AzureAD cmdlets. However, you can only manage mail-related properties by using Exchange Online PowerShell.

Note: At the time of writing this course, the Exchange Online PowerShell V2 (EXO v2) module is the current version. When researching how to connect to Exchange Online, instructions indicating that you need to use PowerShell remoting or the **Connect-EXOPSSession** cmdlet are outdated and shouldn't be followed.

The EXO v2 module includes all of the original cmdlets for managing Exchange Online, and several additional cmdlets that include **EXO** in the cmdlet name. These EXO cmdlets, such as **Get-EXOMailbox**, are more efficient than the original cmdlets.

Installing the EXO v2 module

The EXO v2 module is supported in Windows PowerShell 5.1 and PowerShell 7. Because it's supported in PowerShell 7, it's considered multiplatform. You can use the EXO v2 module in Windows, macOS, and Linux.

To install the EXO v2 module, run the following command:

```
Install-Module -Name ExchangeOnlineManagement
```

Preparing to connect

To use the EXO v2 module, you need to allow scripts. You can set the execution policy to **RemoteSigned** or **Unrestricted**. If you don't allow scripts, you'll notice the error **Files cannot be loaded because running scripts is disabled on this system**.

You also need to allow basic authentication for the WinRM client. This is enabled by default in Windows 10, but some organizations have disabled basic authentication for WinRM as part of security hardening. If Basic authentication isn't enabled, you'll notice the error **The WinRM client cannot process the request. Basic authentication is currently disabled in the client configuration**.

To review the authentication configuration for the Windows Remote Management (WinRM) client, run the following command:

```
winrm get winrm/config/client/auth
```

To enable basic authentication for the WinRM client, run the following command:

```
winrm set winrm/config/client/auth '@{Basic="true"}'
```

Note: If you run this command from a command prompt instead of a PowerShell prompt, don't include the single quotes around `@{Basic="true"}`.

Even though you need to enable Basic authentication in the WinRM client, the EXO v2 module authenticates to Exchange Online by using Modern authentication. In some rare cases, Modern authentication might not be enabled for Exchange Online and you'll need to enable it.

Note: All Exchange Online deployments should be using Modern authentication. This is because it has significant security enhancements over Basic authentication.

Connecting to Exchange Online

You can connect to Exchange Online by using the **Connect-ExchangeOnline** cmdlet with no additional parameters. When you connect to Exchange Online, you're prompted for a username and password to sign in. You need to sign in with a user account that has sufficient privileges to complete the actions you want to perform. You might also be prompted for multifactor authentication.

Note: For tenants in China or Germany, you need to use the *-ExchangeEnvironmentName* parameter and specify **O365China** or **O365Germany**, respectively. For secure US government tenants, you need to specify **O365USGovGCCHigh** or **O365USGovDOD**.

If you're behind a proxy server, you might need to provide proxy options as part of connecting. To do this, provide a **PSSessionOption** object the proxy configuration information. The following example depicts how to create a new **PSSessionOption** object and then use it when connecting to Exchange Online:

```
$ProxyOptions = New-PSSessionOption -ProxyAccessType IEConfig
Connect-ExchangeOnline -PsSessionOption $ProxyOptions
```

Managing mailboxes in Exchange Online

Mailboxes are created automatically for users who are assigned a license that includes an Exchange Online service plan. As such, there's no need to manually create mailboxes for users. Mailboxes are also deleted automatically when the license is removed or the Exchange Online service plan is disabled.

You can also create specialized mailboxes such as:

- Room mailboxes. These are scheduled when you book meetings.
- Equipment mailboxes. These are scheduled to help ensure that users have access to equipment such as cars or portable display units.

- Shared mailboxes. These are used for generic email addresses, such as `info@adatum.com`, where multiple users need access to the mailbox and respond to the messages.

Creating mailboxes

When you use the **New-Mailbox** cmdlet to create a mailbox, it creates a user account at the same time. For resource mailboxes and shared mailboxes, the user account is disabled and doesn't require a license.

When you create one of these mailboxes, you only need to indicate which type of mailbox you're creating and the name of the mailbox. The following example creates a room mailbox:

```
New-Mailbox -Room -Name BoardRoom
```

After creating a resource or shared mailbox, you still need configured permissions. By default, no one has access to those mailboxes. Configuring permissions is covered later in this topic. Configuring calendar booking for resources is covered in the next unit, **Managing resources in Exchange Online**.

Modifying mailboxes

To modify a mailbox's configuration, you use the **Set-Mailbox** cmdlet. There are some mailbox properties that you can configure using **Set-Mailbox**, which you can't configure using the web-based administrative tool. When you review the help information for **Set-Mailbox**, pay careful attention to the parameter descriptions. Some parameters aren't available for managing mailboxes in Exchange Online.

The following table lists some of the parameters for **Set-Mailbox**.

Table 1: Parameters for mailbox configuration

Parameter	Description
-AuditDelegate	Specifies actions on a mailbox that are audited when a delegate performs them, such as **SendOn-Behalf** or **UpdateInboxRules**.
-AuditEnabled	Turns on auditing for a mailbox. This is disabled by default.
-AuditOwner	Specifies actions on a mailbox that are audited when the user performs them, such as **SendOnBe-half** or **UpdateInboxRules**.
-DeliverToMailboxAndForward	When a forwarding SMTP address is configured and this parameter is `$true`, this parameter configures the mailbox to both retain and forward a copy of the messages.
-EmailAddresses	Configures email addresses for a mailbox. The email addresses are stored as an array and typically start with **smtp:**. The primary email address will have the prefix capitalized as **SMTP:**.
-ForwardingSmtpAddress	Specifies an SMTP address for forwarding. To stop forwarding messages, set this value to `$null`.
-GroupMailbox	Required to modify the mailbox associated with a Microsoft 365 group.
-HiddenFromAddressListsEnabled	Specifies whether the mailbox is available in address lists.

Parameter	Description
-MailboxRegion	Specifies the geographic region in which the mailbox should be stored. Used by organizations with a worldwide presence.
-Type	Changes the type of mailbox. Specifies whether a mailbox is regular or used for a special purpose. Special-purpose mailboxes include both shared and resource mailboxes.

The following syntax configures forwarding on a mailbox:

```
Set-Mailbox AbbieP@adatum.com -ForwardingSmtpAddress DoraM@adatum.com
-DeliverToMailboxAndForward $true
```

Querying mailboxes in Exchange Online

To query a list of mailboxes, you can use the **Get-Mailbox** or **Get-EXOMailbox** cmdlets. The primary difference between them is how the data is returned. The **Get-Mailbox** cmdlet returns all properties for the mailboxes. The **Get-EXOMailbox** cmdlet returns only a small set of properties, although you can specify additional properties. This makes **Get-EXOMailbox** much more efficient when working with large data sets.

To obtain additional properties when using the **Get-EXOMailbox** cmdlet, you can use either the *-Properties* parameter or the *-PropertySets* parameter. When using the *-Properties* parameter, you provide a list of properties to return. When you use the *-PropertySets* parameter, you provide a list of predefined property groups that pertain to a specific category. Some property sets that you can specify are:

- All
- Minimum (default value)
- Audit
- Delivery
- Moderation
- Resource

Both cmdlets support using the *-Filter* parameter to select mailboxes matching specific criteria. There are also additional specific parameters that you can use. The following table list some parameters that are available for both cmdlets.

Table 2: Parameters for the Get-Mailbox and Get-EXOMailbox cmdlets

Parameter	Description
-Archive	Returns mailboxes with an archive enabled.
-GroupMailbox	Returns only mailboxes associated with Microsoft 365 groups.
-Identity	Identifies a specific mailbox to return properties for.
-RecipientTypeDetails	Returns mailboxes of a specific type such as UserMailbox, TeamMailbox, or RoomMailbox.
-SoftDeletedMailbox	Returns soft-deleted mailboxes that are still available for recovery.

The following syntax queries all of the room mailboxes and returns resource-related properties:

```
Get-EXOMailbox -RecipientTypeDetails RoomMailbox -PropertySets Resource
```

Managing mailbox permissions

You can configure permissions to provide users with access to other mailboxes or individual folders within a mailbox. For example, you might want to give users full mailbox permission to a shared mailbox. Or you might want to change the default permissions assigned to the **Calendar** folder of a specific user mailbox. The following table lists cmdlets that you can use to manage mailbox and mailbox folder permissions.

Table 3: Cmdlets for managing mailbox and mailbox folder permissions

Cmdlet	Description
Add-MailboxPermission	Adds permissions for a user to a mailbox.
Get-MailboxPermission	Lists user permissions that are assigned to a mailbox.
Remove-MailboxPermission	Removes a user's permissions assignment from a mailbox.
Get-EXOMailboxPermission	Lists user permissions that are assigned to a mailbox.
Add-MailboxFolderPermission	Adds permissions for a user to a folder in a mailbox.
Get-MailboxFolderPermission	Lists user permissions that are assigned to a folder in a mailbox.
Remove-MailboxFolderPermission	Removes a user's permissions assignment from a folder in a mailbox.
Set-MailboxFolderPermission	Sets permissions on a folder in a mailbox and overwrites all exiting permissions.
Get-EXOMailboxFolderPermission	Lists user permissions that are assigned to folder in a mailbox.

The following example assigns full mailbox permissions for a user to the **Info** shared mailbox:

```
Add-MailboxPermission -Identity Info -User AbbieP@adatum.com -AccessRights
FullAccess -InheritanceType All
```

Managing resources in Exchange Online

Room and equipment mailboxes are known as *resource mailboxes*. The primary purpose of resource mailboxes is to allow bookings and ensure that the resource isn't scheduled for multiple events or users simultaneously. You configure the scheduling process for resource mailboxes by using the **Set-Calendar-Processing** cmdlet. You can use the **Get-CalendarProcessing** cmdlet to review the current configuration.

Delegates

One of the options for managing resource scheduling is delegates. *Delegates* are users that can accept or reject booking attempts for the resource. For example, if a room resource is included in a meeting request, the delegate receives a message asking to allow or deny the request. The following example depicts a user being configured as a delegate for a room mailbox:

```
Set-CalendarProcessing -Identity BoardRoom -ResourceDelegates AbbieP@
adatum.com
```

Automated booking

Most organizations want to automate the resource booking process so that delegates only need to mediate conflicts. The *-AutomateProcessing* parameter set to **AutoAccept** is used to indicate that booking should be automated. However, there are other parameters that you can use to define when automated booking is allowed. The following table lists parameters that you can use to control automated resource booking.

Table 1: Parameters to control automated booking of resources

Parameter	Description
-AllBookInPolicy	When set to `$true`, requests that meet booking rules are automatically accepted.
-AllowConflicts	When a resource is booked for a recurring meeting request, you can define whether the entire recurring series is declined whenever there are conflicts. You use this parameter with the -ConflictPercentageAllowed or -MaximumConflictInstances parameters.
-AllRequestInPolicy	When set to `$true`, all users are allowed to submit requests that meet the rules. The default configuration is `$false`.
-AllRequestOutOfPolicy	When set to `$true`, all users are allowed to submit requests that don't meet the rules. The default configuration is `$false`.
-AutomateProcessing	The default value of **AutoAccept** allows requests to be accepted automatically. A value of **AutoUpdate** marks requests as tentative and require a delegate to approve them. A value of **None** means no action is taken until a delegate approves or denies the request.
-BookInPolicy	Specifies users or groups for which bookings that meet the rules are automatically accepted.
-EnforceCapacity	When enabled, the capacity configured for the room is enforced.
-MaximumDurationInMinutes	Specifies the maximum allowed duration for meetings.
-RequestInPolicy	Specifies users or group that can submit requests that meet the booking rules.
-RequestOutOfPolicy	Specifies users or group that can submit requests that don't meeting the booking rules.
-ScheduleOnlyDuringWorkHours	When enabled, requests outside of work hours don't meet the booking rules.

Managing admin roles in Exchange Online

Just as there are varying roles that you can use to control management permissions for Microsoft 365, there are roles in Exchange Online as well. In Exchange Online, the preconfigured management roles are referred to a role group, because a group has been assigned the permissions. The following table lists some of the default role groups.

Table 1: Role groups in Exchange Online

Role group	Description
Organization Management	Performs all Exchange Online management tasks.
Recipient Management	Manages recipients such as mailboxes and distribution groups.
View-only Management	Reviews the configuration of all Exchange Online components but doesn't modify them.
Records Management	Manages retention, journaling, and transport rules.
Discovery Management	Manages legal holds and mailbox searches.

You can review information about role groups by using the **Get-RoleGroup** cmdlet. You can modify membership in role groups by using the **Add-RoleGroupMember** and **Remove-RoleGroupMember** cmdlets. The following example depicts how to add a user to the Recipient Management role group:

```
Add-RoleGroupMember -Identity "Recipient Management" -Member AbbieP@adatum.
com
```

The default roles groups are sufficient for many organizations. However, you can create customized role groups that allow you to define granular permissions, down to specific cmdlets that users are allowed to run. You can also define scopes that control which users or groups that administrators are allowed to manage.

Additional reading: For more detailed information about role groups and permissions in Exchange Online, refer to **Permissions in Exchange Online**[5].

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

Which cmdlet can you use to connect to Exchange Online and manage mailboxes?

- ☐ `Connect-MicrosoftTeams`
- ☐ `Connect-SPOTenant`
- ☐ `Connect-ExchangeOnline`
- ☐ `Connect-MSOnline`

5 https://aka.ms/permissions-in-exchange-online

Question 2

Which cmdlet should you use to configure autoaccept settings for a resource mailbox?

☐ `Set-Mailbox`

☐ `Set-CalendarProcessing`

☐ `Set-AzureADUser`

☐ `Set-MailboxFolderPermission`

Manage SharePoint Online with PowerShell

Lesson overview

SharePoint Online is a collaboration service that allows you to store and share information through a web-based interface. Many organizations use SharePoint sites to build web portals for departments. To provide users with access to SharePoint Online, you need to understand how permissions are assigned. Additionally, you need to understand how external sharing is configured to help ensure that sensitive information isn't shared when it shouldn't be.

Lesson objectives

After completing this lesson, you'll be able to:

- Explain how to connect to SharePoint Online by using PowerShell.
- Explain how to manage SharePoint Online users and groups with PowerShell.
- Explain how to manage sites with PowerShell.
- Explain how to manage external user sharing with PowerShell.

SharePoint Online Management Shell overview

SharePoint Online is the cloud-based version of SharePoint that's included as part of Microsoft 365. You can use SharePoint Online to create sites for user collaboration and file storage. However, SharePoint Online is also used as storage for other Microsoft 365 services. For example, Microsoft Teams use SharePoint Online as a location for file storage.

You can manage many SharePoint Online features by using the web-based SharePoint admin center. However, it doesn't provide full management access to SharePoint Online. For some advanced operations, you need to use SharePoint Online Management Shell. The *SharePoint Online Management Shell* is a Windows PowerShell module that you can install.

Installing SharePoint Online Management Shell

You install the SharePoint Online Management Shell from the PowerShell Gallery by running the following command:

```
Install-Module -Name Microsoft.Online.SharePoint.PowerShell
```

The SharePoint Online Management Shell doesn't update automatically. To update the module, run the following command:

```
Update-Module -Name Microsoft.Online.SharePoint.PowerShell
```

Connecting to SharePoint Online

All of the cmdlet nouns in the SharePoint Online Management Shell begin with **SPO**. You can connect to SharePoint Online by using the **Connect-SPOService** cmdlet as in the following example:

```
Connect-SPOService -Url https://adatum-admin.sharepoint.com
```

When you connect to SharePoint Online, you need to provide the URL for your SharePoint Online instance. This URL is based on your Microsoft 365 tenant name. For example, if your Microsoft 365 tenant name is **adatum.onmicrosoft.com**, then the URL for administering SharePoint Online is **adatum-admin.sharepoint.com**. You can also find this URL when you sign in to SharePoint admin center.

Note: To sign in by using **Connect-SPOService**, you need a user account with either SharePoint Admin or Global Administrator permissions for the Microsoft 365 tenant.

Managing SharePoint Online users and groups with PowerShell

To provide access to a SharePoint site, you can assign users and groups varying levels of site permissions. Some of the permissions available by default are:

- Full Control
- Design
- Edit
- Approve
- Contribute
- Read

Note: You can create customized permission levels by using SharePoint admin center. There are no cmdlets in SharePoint Online Management Shell for creating or modifying customized permission levels.

SharePoint groups

To assign permissions in SharePoint Online by using PowerShell, you create SharePoint groups. SharePoint groups exist only in SharePoint Online and are specific to the site in which you create them. The following example depicts how to use the **New-SPOSiteGroup** cmdlet to create a SharePoint Online group and assign permissions for a site:

```
New-SPOSiteGroup -Group MarketingUsers -PermissionLevels Read -Site
https://adatum.sharepoint.com/sites/Marketing
```

You can use the **Get-SPOSiteGroup** cmdlet to identify the SharePoint groups that have been created for a site and assigned permissions. The results also contain the group membership. You need to specify the site URL with the request as depicted in the following example:

```
Get-SPOSiteGroup -Site https://adatum.sharepoint.com/sites/Marketing
```

You can modify the permissions assigned to a SharePoint group by using the **Set-SPOSiteGroup** cmdlet. You need to specify the name of the group and the site URL in addition to the permissions to be modified. To add permissions, use the *-PermissionLevelsToAdd* parameter. To remove permissions, use the *-PermissionLevelsToRemove* parameter. The following example uses the *-PermissionLevelsToAdd* parameter:

```
Set-SPOSiteGroup -Site https://adatum.sharepoint.com/sites/Marketing -Group
MarketingUsers -PermissionLevelsToAdd Contribute
```

Managing site users

To give permissions to Azure AD users, you must make them members of a SharePoint group. You can add members to a SharePoint group by using the **Add-SPOUser** cmdlet as depicted in the following example. You need to specify the site URL along with the group name:

```
Add-SPOUser -Site https://adatum.sharepoint.com/sites/Marketing -Group
MarketingUsers -LoginName AbbieP@adatum.com
```

Note: You can also add security groups from Azure AD as members of SharePoint groups by using the **Add-SPOUser** cmdlet.

The **Remove-SPOUser** cmdlet has similar syntax to the **Add-SPOUser** cmdlet but removes a user from the specified SharePoint group. If you don't specify a group, then the user is removed from all SharePoint groups in the site.

To review the users who are members of SharePoint groups, you can use the **Get-SPOUser** cmdlet. You need to specify the site URL from which to retrieve users, but the group is optional. If you don't specify the group, you'll receive a list of all users in the site and the SharePoint groups that they are members of.

There is also a **Set-SPOUser** cmdlet but its only purpose is to set whether a user is an administrator for the site. The following example depicts how to configure a user as a site administrator:

```
Set-SPOUser -Site https://adatum.sharepoint.com/sites/Marketing -LoginName
AbbieP@adatum.com -IsSiteCollectionAdmin $true
```

Managing SharePoint sites with Windows PowerShell

In SharePoint Online, you can have multiple sites that contain different content. There's a default site designed for collaboration that's created automatically when you create your tenant. There are also sites created automatically for each Microsoft 365 group or Microsoft Team and OneDrive users. Additionally, you can create your own SharePoint sites and customize them.

Creating sites

You can use the **New-SPOSite** cmdlet to create new sites in SharePoint online. When you create a new site, the parameters in the following table are required.

Table 1: Parameters for creating a new site

Parameter	Description
-Url	The URL for the site, which needs to be with your SharePoint Online namespace. For example, if the default URL for your SharePoint Online tenant is `https://adatum.sharepoint.com`, then the URL for the site could be `https://adatum.sharepoint.com/sites/Marketing`.
-Owner	The owner of the site that can manage it.
-StorageQuota	The maximum size of the site in megabytes (MB). This must be less than the quota available in the tenant.

The following example creates a new site with `AbbieP@adatum.com` as the owner and a 256 MB storage quota:

```
New-SPOSite -Url https://adatum.sharepoint.com/sites/Marketing -Owner
AbbieP@adatum.com -StorageQuota 256
```

Most of the time when you create a site, you'll want to base it on a template. A template defines components that are automatically included in a site. There are templates included in SharePoint Online by default and you can also make your own. To review the template in your SharePoint Online tenant, use the **Get-SPOWebTemplate** cmdlet. To use a template when creating a site with the **New-SPOSite** cmdlet, use the *-Template* parameter.

Modifying sites

You can use the **Set-SPOSite** cmdlet to modify existing sites. To define which site you want to modify, use the *-Identity* parameter and provide the URL for the site. The URL for a site is a unique identifier for the site, and you cannot modify it by using **Set-SPOSite**. The following table lists some parameters you can use with **Set-SPOSite**.

Table 2: Parameters for modifying existing sites

Parameter	Description
-Title	Sets the title for the site. The title typically displays when users sign in to the site. For example, a title might be **Marketing Portal**.
-StorageQuotaWarningLevel	Sends a warning message to the site owner when the warning level is reached. This value should be less than the -StorageQuota.
-AllowEditing	Controls whether users are allowed to edit Office files in the browser, and copy and paste Office file content out of the browser window.
-LockState	Sets the lock state on a site. Valid values are: **NoAccess**, **ReadOnly**, and **Unlock**.

The following example sets the title for a site:

```
Set-SPOSite -Identity https://adatum.sharepoint.com/sites/Marketing -Title
"Marketing Portal"
```

Listing and reviewing sites

You can use the **Get-SPOSite** cmdlet to review the sites created in your SharePoint Online tenant and their configurations. To list all sites in your tenant, don't include any parameters. To list the specific site's properties, you use the *-Identity* parameter and specify the site's URL. The following example depicts how to list a specific site's properties:

```
Get-SPOSite -Identity https://adatum.sharepoint.com/sites/Marketing |
Format-List
```

Removing sites

You use **Remove-SPOSite** to remove a site. The following example depicts how to use this cmdlet to remove a site:

```
Remove-SPOSite -Identity https://adatum.sharepoint.com/sites/Marketing
```

When you remove a site, it's placed in the SharePoint Recycle Bin. You can use the **Restore-SPODeleted-Site** cmdlet to restore a site from the SharePoint Recycle Bin. To purge a deleted site from the SharePoint Recycle Bin, you can use the **Remove-SPODeletedSite** cmdlet.

SharePoint Online terminology

The SharePoint admin center uses the site and sub-site terminology to describe how to nest content. When you create a sub-site, it inherits some of the settings from the site, such as the owner.

If you're familiar with the on-premises version of SharePoint, it uses the terminology *site collections* and *sites* instead. When reviewing documentation for SharePoint Online cmdlets, you might notice that it often uses the on-premises site collections terminology. There are no cmdlets for managing sites within site collections (or sub-sites).

Managing external user sharing with Windows PowerShell

SharePoint Online content can be shared with external users. Because Microsoft OneDrive storage is part of SharePoint Online, there are similar settings for both. When you configure settings for external sharing, the OneDrive settings must be the same or more restrictive than the SharePoint Online settings. The commonly used external sharing settings for SharePoint Online are listed in the following table.

Table 1: Commonly used external sharing settings for SharePoint Online

Permission level	Description
Anyone	Allows users to share files and folders by using links that let anyone who has the link access the files or folders without authenticating. This setting also allows users to share sites with new and existing guests who authenticate. If you select this setting, you can restrict the **Anyone** links so that they expire within a specific number of days, or so that they provide only **View** permission.
New and existing guests	Requires people who have received invitations to sign in with their work or school account (if their organization uses Microsoft 365). Alternatively, they can use a Microsoft account, or provide a code to verify their identity. Users can share with guests already in your organization's directory, and they can send invitations to people who will be added to the directory if they sign in.

Permission level	Description
Existing guests	Allows sharing only with guests who are already in the directory. These guests might exist in your directory because they previously accepted sharing invitations or because they were added manually, such as through Azure business-to-business (B2B) collaboration.
Only people in your organization	Doesn't allow external sharing.

Managing sharing

To configure these permissions for a site, you use the **Set-SPOSite** cmdlet with the *-SharingCapability* parameter. Valid values for the *-SharingCapability* parameter are:

- **ExternalUserAndGuestSharing**
- **ExternalUserSharingOnly**
- **ExistingExternalUserSharingOnly**
- **Disabled**

The following example disables external sharing for a site:

```
Set-SPOSite -https://adatum.sharepoint.com/sites/Marketing -SharingCapabil-
ity Disabled
```

When you allow sharing with external users, you can restrict sharing based on the user domain by using the *-SharingDomainRestrictionMode* parameter. The following table describes the valid values.

Table 2: Values to restrict sharing based on the user domain

Value	Description
None	Doesn't restrict sharing by domain (default).
AllowList	Allows sharing only with external users that have an account on domains specified by using the -SharingAllowedDomainList parameter.
BlockList	Allows Sharing with external users in all domains except in domains specified by using the -Sharing-BlockedDomainList parameter.

Managing sharing links

When a user shares content from a SharePoint site, a sharing link is sent to the recipient. This link is unique and includes information, such as permissions, to edit the file and who can use it.

You can use the *-DefaultLinkSharingType* parameter to specify the default value for the users who can use the sharing link. Users that are sharing can still select the option that they prefer. The following table lists the valid values.

Table 3: Values to specify which users can use the sharing link

Value	Description
None	Uses the value set at the tenant level.

Value	Description
AnonymousAccess	Sets the default sharing link for this site to an **Anonymous Access** or **Anyone** link.
Internal	Sets the default sharing link for this site to the **organization** link or company shareable link.
Direct	Sets the default sharing link for this site to the **Specific people** link.

You can use the *-DefaultLinkPermission* parameter to specify the default value for what users can do to the content via the sharing link. Users that are sharing can still select the option that they prefer. The following table lists the valid values.

Table 4: Values to specify what users can do to the content through the sharing link

Value	Description
None	Uses the value set at the tenant level.
View	Set the default value to **View** permissions.
Edit	Sets the default value to **Edit** permissions.

You can configure an expiration time for external sharing links. After a link expires, it can no longer be used to access the linked content. The expiration time can be set independently for anonymous and external users. The following table lists parameters that you can use to configure link expiration for a site.

Table 5: Parameters to configure link expiration for a site

Parameter	Description
-OverrideTenantAnonymousLinkExpirationPolicy	To set an expiration time for anonymous links at the site level, set this value to `$true`. When set to `$false`, anonymous link expiration settings from the tenant level are used.
-AnonymousLinkExpirationInDays	This sets the number of days that an anonymous link is valid.
-OverrideTenantExternalUserExpirationPolicy	To set an expiration time for external user links at the site level, set this value to `$true`. When set to `$false`, external user link expiration settings from the tenant level are used.
-ExternalUserExpirationInDays	This sets the number of days that an external user link is valid.

Note: You can configure some sharing settings at the tenant level by using the **Set-SPOTenant** cmdlet.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

When you modify a SharePoint Online site with `Set-SPOSite`*, what do you need to provide to identify which site is being modified?*

- ☐ The URL of the site
- ☐ The display name of the site
- ☐ The group ID of the site
- ☐ The owner of the site

Question 2

You use the `Add-SPOUser` *cmdlet to directly assign permissions for a site. True or False?*

- ☐ True
- ☐ False

Manage Microsoft Teams with PowerShell

Lesson overview

Microsoft Teams is a collaboration service that combines multiple Microsoft 365 services into a single interface. It's a user-friendly service that users with limited training can manage. As a result, it's a useful tool for workgroups and projects. You can use the Microsoft Teams PowerShell module to perform tasks such as creating teams and managing user permissions.

Lesson objectives

After completing this lesson, you'll be able to:

- Describe the Microsoft Teams PowerShell module.
- Explain how to connect to Microsoft Teams by using PowerShell.
- Manage Microsoft Teams with the Microsoft Teams PowerShell module.

Overview of the Microsoft Teams PowerShell module

Microsoft Teams combines many elements of Microsoft 365—such as Microsoft 365 groups, Exchange Online, and SharePoint storage—into a single location for collaboration. You can install the Microsoft Teams module to manage Microsoft Teams by using Windows PowerShell.

You can create and configure Microsoft Teams by using cmdlets with a noun that starts with **Team**, such as the following cmdlets:

- **Get-Team**
- **Add-TeamUser**
- **New-TeamsApp**

The Microsoft Teams module also includes many functions. These functions have the same verb-noun naming format as other cmdlets, but with a noun that starts with **CsTeam**. For example, the following cmdlets are included in the Microsoft Teams module:

- **Set-CsTeamsMeetingPolicy**
- **Remove-CsTeamTemplate**
- **New-CsTeamsEmergencyCallingPolicy**
- **Get-CsTeamsMessagingPolicy**

The functions in the Microsoft Teams module are for configuring of the overall service, but not individual Teams. Also included are commands that you can use to create and configure user policies, and to control and manage communications.

Installing the Microsoft Teams PowerShell module

You can install the Microsoft Teams module from the PowerShell Gallery. To install the Microsoft Teams module, run the following command:

```
Install-Module -Name MicrosoftTeams
```

The Microsoft Teams module doesn't update automatically. To update the SharePoint Online Management Shell, run the following command:

```
Update-Module -Name MicrosoftTeams
```

Connecting to Microsoft Teams

You can connect to Microsoft Teams by using the **Connect-MicrosoftTeams** cmdlet with no additional parameters. When you connect to Microsoft Teams, you're prompted for a username and password to sign in. You might also be prompted for multifactor authentication. Be sure to sign in with a user account that has sufficient privileges to perform the actions you want to complete.

Managing Teams with the Microsoft Teams PowerShell module

Microsoft Teams is a collaboration tool. Within a Microsoft Team, you can create multiple channels to organize data and apps. When there are multiple channels, you can restrict access to channels based on the user. You can use the Microsoft Teams PowerShell module to create, configure, and manage teams, user settings, and channels.

Creating teams

To create a new team, you use the **New-Team** cmdlet as depicted in the following example:

```
New-Team -DisplayName "Marketing Team"
```

When you create a new team by using PowerShell, you can't specify a template unless you're an education customer. When you create a team from within the Microsoft Teams client, you can specify a template or copy an existing team. You have the same options when creating a team by using Graph API. To get a list of available templates, use the **Get-CsTeamTemplateList** cmdlet. You can also create your own templates.

Note: You can convert an existing Microsoft 365 group to a team by using the *-GroupId* parameter when creating a new team.

If you don't specify an owner when you create a team, then by default you become the owner of the team. If your administrative user doesn't have a Microsoft Teams license, you need to specify an owner with a Microsoft Teams license. Otherwise, the team creation fails.

Configuring teams

When you create a new team, a new Microsoft 365 group is created as part of the team. When you manage an existing team, you need to refer to the Microsoft 365 group ID as the unique identifier for the team. The group ID displays when you create the group. You can also obtain the group ID by using the **Get-Team** cmdlet.

You can modify team settings by using the **Set-Team** cmdlet. Some parameters that you can use are listed in the following table.

Table 1: Parameters for modifying team settings

Parameter	Description
-Description	Provides a description of the team (1,024 characters or less) to make it easier for users to identify the team's purpose.
-MailNickName	Specifies the alias for the associated Microsoft 365 group that's used when creating the **PrimarySmtpAddress**.
-Visibility	Determines whether the team is public or private. Public teams are noticeable to everyone in the team gallery, and anyone can join without team owner approval. Whereas private teams can only be joined if the team owner adds someone to them.
-AllowAddRemoveApps	Determines whether or not members (and not just owners) are allowed to add apps to the team.
-AllowCreateUpdateChannels	Determines whether or not members (and not just owners) are allowed to create channels.
-AllowUserEditMessages	Determines whether or not users can edit messages that they've posted.

You can also use these same parameters when you create the team. The following example depicts how to specify the mail nickname:

```
Set-Team -GroupId 26be526d-201a-4af6-9918-2fdbf6306916 -MailNickName "Mar-
ketingTeam"
```

Managing team members

After you create a team, you can manage team members by using the **Add-TeamUser** and **Remove-TeamUser** cmdlets. You can add users as members or owners. When you use these cmdlets, you need to specify the **GroupId** associated with the team as in the following example:

```
Add-TeamUser -GroupId 26be526d-201a-4af6-9918-2fdbf6306916 -User AbbieP@
adatum.com -Role Member
```

Creating and configuring team channels

A team can have multiple channels that contain content. The following table depicts some of the cmdlets that you can use to create and manage channels.

Table 2: Cmdlets for creating and managing channels

Cmdlet	Description
New-TeamChannel	Creates a new channel in a team.
Get-TeamChannel	Lists the channels in a set.
Set-TeamChannel	Modifies the display name or description for a channel.
Add-TeamChannelUser	Adds a user as a member or owner of a channel.
Remove-TeamChannelUser	Removes a user from a channel.

Note: At the time of writing this course, the **Add-TeamChannelUser** and **Remove-TeamChannelUser** cmdlets are available only in the preview release of the Microsoft Teams module.

Test your knowledge

Use the following questions to check what you've learned in this lesson.

Question 1

When you use the `Connect-MicrosoftTeams` *cmdlet, you don't need to provide any additional parameters. True or False?*

☐ True

☐ False

Question 2

Which parameter can you use with the `Set-MicrosoftTeam` *cmdlet to set the primary SMTP address for the team?*

☐ `-DisplayName`

☐ `-MailNickName`

☐ `-Visibility`

☐ `-User`

Module 10 lab and review

Lab: Managing Microsoft 365 with PowerShell

Scenario

You've created a new Microsoft 365 tenant. As a new administrator, you want to try using PowerShell to manage some of the Microsoft 365 services before you start deploying them to users.

Objectives

After completing this lab, you'll be able to:

- Manage users in Azure AD.
- Manage Exchange Online.
- Manage SharePoint Online.
- Manage Microsoft Teams.

Estimated time: 60 minutes

Module review

Use the following questions to check what you've learned in this module.

Question 1

When you use the `Set-Team` *cmdlet to modify a team, what is the unique identifier that you need to specify?*

☐ Display name

☐ Team ID

☐ Group ID

☐ URL

Question 2

Which modules can you use to connect to Azure AD? (Select two.)

☐ ExchangeOnlineManagement

☐ Microsoft.Online.SharePoint.PowerShell

☐ MicrosoftOnline

☐ MicrosoftTeams

☐ AzureAD

Question 3

When you configure a user license by using cmdlets in the Azure AD PowerShell for Graph module, which object types do you need to create? (Select two.)

☐ RemoveLicenses

☐ AddLicenses

☐ AssignedLicenses

☐ AssignedLicense

☐ DisabledPlans

Question 4

When you use the `Connect-SPOService` *cmdlet, you don't need to provide any additional parameters. True or False?*

☐ True

☐ False

Question 5

Which cmdlet allows you to specify property sets that limit the amount of data returned?

☐ `Get-EXOMailbox`

☐ `Get-Mailbox`

☐ `Get-AzureADUser`

☐ `Get-MsolUser`

Answers

Question 1

Which cmdlet can you use to connect to Azure AD to create users?

☐ `Connect-MicrosoftTeams`

☐ `Connect-SPOTenant`

☐ `Connect-ExchangeOnline`

■ `Connect-MSOnline`

Explanation
`Connect-MSOnline` *is the correct answer. After you install the Azure Active Directory Module for Windows PowerShell module, you can use* `Connect-MSOnline` *to connect to Azure AD to manage users, groups, and licensing. Alternatively, you can use* `Connect-AzureAD` *if you installed the Azure AD PowerShell for Graph module instead.*

Question 2

Which role should be assigned to a user to manage all services in Microsoft 365?

■ Global Administrator

☐ Global Reader

☐ Service Support Admin

☐ User Administrator

Explanation
Global Administrator is the correct answer. Only the Global Administrator role provides permissions to manage all services in Microsoft 365. Other roles provide permissions to manage a subset of Microsoft 365 services.

Question 1

Which cmdlet can you use to connect to Exchange Online and manage mailboxes?

☐ `Connect-MicrosoftTeams`

☐ `Connect-SPOTenant`

■ `Connect-ExchangeOnline`

☐ `Connect-MSOnline`

Explanation
`Connect-ExchangeOnline` *is the correct answer. After you install the ExchangeManagement module, you can use* `Connect-ExchangeOnline` *to connect to Exchange Online and manage mailboxes.*

Question 2

Which cmdlet should you use to configure autoaccept settings for a resource mailbox?

☐ `Set-Mailbox`

■ `Set-CalendarProcessing`

☐ `Set-AzureADUser`

☐ `Set-MailboxFolderPermission`

Explanation
`Set-CalendarProcessing` *is the correct answer. After you create a resource mailbox, you can use the* `Set-CalendarProcessing` *cmdlet to configure the settings that control when meeting requests are automatically accepted.*

Question 1

When you modify a SharePoint Online site with `Set-SPOSite`, what do you need to provide to identify which site is being modified?

■ The URL of the site

☐ The display name of the site

☐ The group ID of the site

☐ The owner of the site

Explanation
The URL of the site is the correct answer. The URL for each SharePoint Online site is a unique identifier that's provided with the `–Site` *parameter when running the* `Set-SPOSite` *cmdlet.*

Question 2

You use the `Add-SPOUser` cmdlet to directly assign permissions for a site. True or False?

☐ True

■ False

Explanation
False is the correct answer. To assign permissions to a SharePoint Online site, you create a SharePoint group that's assigned permissions, and then use the `Add-SPOUser` *cmdlet to add members to the SharePoint group.*

Question 1

When you use the `Connect-MicrosoftTeams` cmdlet, you don't need to provide any additional parameters. True or False?

■ True

☐ False

Explanation
True is the correct answer. When you use the `Connect-MicrosoftTeams` *cmdlet without any additional parameters, you're prompted for credentials, and then connected to Microsoft Teams.*

Question 2

Which parameter can you use with the `Set-MicrosoftTeam` cmdlet to set the primary SMTP address for the team?

- ☐ `-DisplayName`
- ■ `-MailNickName`
- ☐ `-Visibility`
- ☐ `-User`

Explanation
`-MailNickName` *is the correct answer. The mail nickname for a team is used to generate the primary SMTP address for the team.*

Question 1

When you use the `Set-Team` cmdlet to modify a team, what is the unique identifier that you need to specify?

- ☐ Display name
- ☐ Team ID
- ■ Group ID
- ☐ URL

Explanation
Group ID is the correct answer. Each team includes a Microsoft 365 group that has a unique group ID. When modifying a team, you need to specify the group ID of the associated Microsoft 365 group by using the `-GroupId` *parameter.*

Question 2

Which modules can you use to connect to Azure AD? (Select two.)

- ☐ ExchangeOnlineManagement
- ☐ Microsoft.Online.SharePoint.PowerShell
- ■ MicrosoftOnline
- ☐ MicrosoftTeams
- ■ AzureAD

Explanation
The correct answers are MicrosoftOnline and AzureAD. The MicrosoftOnline module provides the `Connect-MSOnline` *cmdlet, and the Azure AD PowerShell for Graph module provides the* `Connect-AzureAD` *cmdlet.*

Question 3

When you configure a user license by using cmdlets in the Azure AD PowerShell for Graph module, which object types do you need to create? (Select two.)

☐ RemoveLicenses

☐ AddLicenses

■ AssignedLicenses

■ AssignedLicense

☐ DisabledPlans

Explanation
The correct answers are AssignedLicenses and AssignedLicense. You need to create both an AssignedLicense object and an AssignedLicenses object. The AssignedLicense object is added to either the RemoveLicenses or AddLicenses property of the AssignedLicenses object. The AssignedLicenses object is assigned to the user.

Question 4

When you use the `Connect-SPOService` cmdlet, you don't need to provide any additional parameters. True or False?

☐ True

■ False

Explanation
False is the correct answer. When you use the `Connect-SPOService` *cmdlet to connect to SharePoint Online, you need to use the* `-Url` *parameter to provide the administrative URL for your SharePoint Online tenant. You'll also be prompted for authentication credentials.*

Question 5

Which cmdlet allows you to specify property sets that limit the amount of data returned?

■ `Get-EXOMailbox`

☐ `Get-Mailbox`

☐ `Get-AzureADUser`

☐ `Get-MsolUser`

Explanation
`Get-EXOMailbox` *is the correct answer. To increase efficiency, the* `Get-EXOMailbox` *cmdlet returns a limited set of data by default. You can use the* `-Properties` *or* `-PropertySets` *parameters to specify additional properties to return.*

Module 11 Using background jobs and scheduled jobs

Use background jobs

Lesson overview

In addition to the traditional tasks that run in the foreground within the Windows PowerShell console or ISE, you also can run background tasks. When you run a command as a background job, Windows PowerShell performs the task asynchronously in its own thread. This thread is separate from the pipeline thread that the command uses. When a command runs as a background job, even if it takes a long time to complete, you regain access to the PowerShell prompt immediately. This allows you to run other commands while the job runs in the background.

In this lesson, you'll learn about three types of jobs: local jobs, Windows PowerShell remote jobs, and Common Information Model (CIM)/Windows Management Instrumentation (WMI) jobs. These three job types form the basis of the Windows PowerShell job system.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the purpose and functionality of background jobs.
- Start jobs.
- Manage jobs.
- Retrieve job results.
- Use background jobs.

What are background jobs?

There are three basic types of background jobs:

- *Local jobs* run their commands on the local computer and typically access only local resources. However, you can create local jobs that target a remote computer. For example, you could create a local job that includes the following command, where the *–ComputerName* parameter makes it connect to a remote computer:

  ```
  Get-Service –Name * -ComputerName LON-DC1
  ```

- *Remote jobs* use Windows PowerShell remote to transmit 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* use the Common Information Model (CIM) and Windows Management Instrumentation (WMI) repository of management information. The commands run on your computer but can connect to one or more remote computer's repository. Local jobs that use CIM commands use the **Start-Job** command, whereas WMI and other commands use the *-AsJob* parameter within a WMI command.

Each job type has specific characteristics. For example, local and Windows PowerShell remote jobs run in a background Windows PowerShell run space. Think of them as running in a hidden instance of Windows PowerShell. Other types of jobs might have different characteristics. Also, add-in modules add more job types to Windows PowerShell, and those job types have their own characteristics. Remote jobs are useful for managing multiple remote computers simultaneously. Remote computers run Windows PowerShell remote commands by using their own local resources. Therefore, you can include any command in the job.

Remember that these aren't the only job types that Windows PowerShell supports. Use modules and other add-ins to create additional job types. Workflow jobs help you automate long-running tasks, or *workflows*, by simultaneously targeting multiple managed computers or devices. Windows PowerShell workflows offer additional resiliency benefits. You can restart target devices and, if a workflow isn't finished, it automatically restarts on the target device or devices and continues with the job's workflow commands.

It's important to note that interactive Windows PowerShell console workflow jobs aren't available in a noninteractive Windows PowerShell console. Scheduling a task to run at machine startup won't find suspended jobs from a noninteractive console. Scheduled tasks can't find workflow jobs to resume unless you're signed in to an interactive session.

Starting jobs

You start each of the three type of jobs, namely local, remote, and Common Information Model (CIM)/ Windows Management Instrumentation (WMI), in a different way. The following sections describe specific methods for 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 receive a sequential job identification (ID) number and a default job name. You can't change the assigned job ID number, but you can use the *–Name* parameter to specify a custom job name. Custom names make it easier to retrieve a job and identify it in the job list.

Note: At first, job ID numbers might not seem to be sequential. However, you'll learn the reason for this later in this module.

You can specify the *–Credential* parameter to run a 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 in other sessions.

Here are examples of how to start local jobs:

```
PS C:\> Start-Job -ScriptBlock { Dir C:\ -Recurse } -Name LocalDirectory

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
2      LocalDirectory  BackgroundJob   Running       True            local-
host


PS C:\> Start-Job -FilePath C:\test.ps1 -Name TestScript

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
4      TestScript      BackgroundJob   Running       True            local-
host
```

Remote jobs

Start Windows PowerShell remote jobs by running **Invoke-Command**. This is the same command that sends commands to a remote computer. Add the *–AsJob* parameter to make the command run in the background. Use the *–JobName* parameter to specify a custom job name. All other parameters of **In-voke-Command** are used in the same way. Here's an 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     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
6      RemoteLogs      RemoteJob       Running       True            LON-
DC1...
```

Note: 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 results.

The computer on which you run **Invoke-Command** creates and manages remote jobs. You can refer to that computer as the *initiating computer*. The commands inside the job are transmitted to remote computers, which then run them and return results to the initiating computer. The initiating computer stores the job's results in its memory.

CIM and WMI jobs

To use CIM commands in a job, you must launch the job with **Start-Job**. Here's an example:

```
PS C:\> Start-Job  -ScriptBlock {Get-CimInstance -ClassName Win32_Computer-
System}

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 as jobs by using **Start-Job**. An example is **Invoke-Cim-Method**.
The fact that CIM commands don't have an *–AsJob* parameter isn't important. 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'd use to query WMI instanc-es. Add the *–AsJob* parameter to run the command on a background thread. There's no option to provide a custom job name. The **Get-Help** information for **Get-WmiObject** states the following for the *–AsJob* parameter:

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 7 and newer versions of Windows.

WMI jobs don't require that you enable Windows PowerShell remoting on either the initiating computer or the remote computer. However, they do require that WMI be accessible on the remote computers.

Here's an example:

```
PS C:\> Get-WmiObject -Class Win32_NTEventLogFile -ComputerName local-
host,LON-DC1 -AsJob

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
10     Job10           WmiJob          Running       True            local-
ho...
```

Job objects

Notice that each of the preceding examples results in a job object. It represents the running job, and you can use it to monitor and manage the job.

Managing jobs

When you start a job, you're given a job object that enables you 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 represents the entire job, regardless 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 and WMI jobs contain one child job for each computer that you specify.

Retrieving jobs

Run **Get-Job** to list all current jobs. You can list a specified job by adding the *–ID* or *–Name* parameter and specifying the desired job ID or job name. Retrieve child jobs by using the job ID. Here are examples:

```
PS C:\> Get-Job

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
2      LocalDirectory  BackgroundJob   Running       True            local-
host
4      TestScript      BackgroundJob   Completed     True            local-
host
6      RemoteLogs      RemoteJob       Failed        True            LON-
DC1...
10     Job10           WmiJob          Failed        False           local-
ho...


PS C:\> Get-Job -Name TestScript

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
4      TestScript      BackgroundJob   Completed     True            local-
host


PS C:\> Get-Job -ID 5

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
```

```
--
5       Job5                            Completed     True            local-
host
```

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 job contains four child jobs, and three of those jobs finished successfully but one failed, the parent job status will be **Failed**.

Listing jobs

List the child jobs of a specified parent job by retrieving the parent job object and expanding its **Child-Jobs** property. In Windows PowerShell 3.0 and newer, use the *-IncludeChildJobs* parameter of **Get-Job** to display a job's child jobs. Here's an example:

```
C:\PS>Get-Job -Name RemoteJobs -IncludeChildJobs

Id      Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--      ----            -------------   -----         -----------     ------
--
7       Job7                            Failed        False           LON-
DC1
8       Job8                            Completed     True            LON-
CL1
9       Job9                            Failed        False           LON-
SVR1
```

The earlier method uses the following example:

```
PS C:\> Get-Job -Name RemoteLogs | Select -ExpandProperty ChildJobs

Id      Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--      ----            -------------   -----         -----------     ------
--
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.

Job-management commands

Manage jobs by using several available Windows PowerShell commands. Pipe one or more jobs to each of these commands or specify jobs by using the *–ID* or *–Name* parameters. Both 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**, which halts a job that's running. Use this command to cancel a job that's in an infinite loop or that has run longer than you want.
- **Remove-Job**, which deletes a job object, including any command results stored in memory. Use this command when you're finished working with a job, so the shell releases memory.
- **Wait-Job**, which you typically use in a script. It pauses script processing until the jobs you indicate reach the specified state. Use this command 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 a PowerShell session, Windows PowerShell removes all jobs and their results, and you can no longer access them.

Retrieving results for running jobs

When a job runs, Windows PowerShell stores all command outputs in memory, starting with the first output that the command produced. You don't have to wait for a command to complete before output becomes available.

The job list indicates whether a job has stored results that you haven't yet retrieved. Here's an example:

```
PS C:\> Get-Job

Id     Name            PSJobTypeName   State         HasMoreData     Loca-
tion
--     ----            -------------   -----         -----------     ------
--
13     Job13           BackgroundJob   Running       True            local-
host
```

In this example, job ID 13 is still running, but the **HasMoreData** column indicates that results already have been stored in memory. To receive a job's results, use the **Receive-Job** command.

By default, job results are removed from memory after they're 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'll receive the results from all child jobs. You also can retrieve the results of a single child job or multiple child jobs.

You also can retrieve the results of a job that's still running. However, unless you specify *–Keep*, the job object's results will be empty until the job's command adds new output.

Here's an example:

```
Receive-Job –ID 13 -Keep | Format-Table –Property Name,Length
```

Demonstration: Using background jobs

In this demonstration, you'll learn how to create and manage local and remoting jobs.

Demonstration steps

1. On **LON-CL1**, open the Windows PowerShell Integrated Scripting Environment (ISE) as **Administrator**.
2. Enter the following command, and then press the Enter key:
   ```
   Enable-PSRemoting
   ```
3. Enter the following command, and then press the Enter key:
   ```
   Start-Job –ScriptBlock { Dir C:\ -Recurse } –Name LocalDir
   ```
4. Enter the following command, and then press the Enter key:
   ```
   Invoke-Command –ScriptBlock { Get-EventLog –LogName Security –Newest 100 } –ComputerName
   LON-CL1,LON-DC1 –JobName RemoteLogs
   ```
5. Enter the following command, and then press the Enter key:
   ```
   Get-Job
   ```
6. Enter the following command, and then press the Enter key:
   ```
   Get-Job –Name LocalDir | Stop-Job
   ```
7. Enter the following command, and then press the Enter key:
   ```
   Receive-Job  –Name LocalDir
   ```
8. Enter the following command, and then press the Enter key:
   ```
   Remove-Job –Name LocalDir
   ```
9. Enter the following command, and then press the Enter key:
   ```
   Get-Job
   ```

Note: Repeat this step until the **RemoteLogs** job returns a status of **Completed**.

1. Enter the following command, and then press the Enter key:
   ```
   Get-Job –Name RemoteLogs -IncludeChildJob | Where location -eq 'LON-DC1' | Select -ExpandProp-
   erty ID
   ```
2. Note the job ID that corresponds to the **LON-DC1** job.
3. Enter the following command, replacing `<id>` with the job ID number from the previous step, and then press the Enter key:

```
Get-Job –ID <id> | Receive-Job –Keep
```

4. Enter the following command, and then press the Enter key:

```
Receive-Job –Name RemoteLogs
```

5. Enter the following command, and then press the Enter key:

```
Remove-Job –Name RemoteLogs
```

6. Leave the Windows PowerShell ISE open for the next demonstration.

Test your knowledge

Use the following question to check what you've learned in this lesson.

Question 1

What are examples of tasks that you might want to run in the background?

Use scheduled jobs

Lesson overview

In this lesson, you'll learn to use scheduled jobs. They're similar to background jobs and run asynchronously in the background. In Windows PowerShell, scheduled jobs are essentially scheduled tasks. They follow all of the same rules for actions, triggers, and other features, and run Windows PowerShell scripts by design.

Lesson objectives

After completing this lesson, you'll 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 and triggers.
- Create scheduled jobs.
- Retrieve scheduled job results.
- Use scheduled jobs.

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. Additionally, like tasks, job results are saved to disk, and scheduled jobs can run even if Windows PowerShell isn't running.

Scheduled tasks

Scheduled tasks are part of the Windows core infrastructure components. Other Windows components and products that run on Windows extensively use scheduled tasks. They're generally simpler than scheduled jobs. The **Task Scheduler** enables you to configure a schedule for running almost any program or process, in any security context, triggered by various system events or a particular date or time.

However, scheduled tasks can't capture and manipulate task output. A scheduled task can run almost anything runnable on a Windows device, so it's 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 runs 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:

- The **Action**, which specifies the program to be run.
- The **Principal**, which identifies the context to use to run an action.
- The **Trigger**, which defines the time or system event that determines when the program is to be run.
- The **Additional settings**, which further configure the task and control how the action is run.

Commands that work with scheduled tasks are in the **ScheduledTasks** module that's included with Windows 10 and Windows Server 2019. To review the complete list of commands, run the following command:

```
Get-Command –Module ScheduledTasks
```

As an example, check on the available scheduled tasks by running the **Get-ScheduledTask** command. This will list all available scheduled tasks, regardless of whether they're enabled or disabled.

Get information on a specific task by running **Get-ScheduledTask** with the *-TaskPath* parameter. For best practices, ensure that you put the actual path in quotes. Get further information about a particular task by using the **Get-ScheduledTaskInfo** command. You can then combine these using a pipeline to get additional information. For example, retrieve detailed information about the **Automatic Update** task running on your system by entering the following command:

```
Get-ScheduledTask -TaskPath "\Microsoft\Windows\WindowsUpdate\" |
Get-ScheduledTaskInfo
```

You also can create and run scheduled tasks from the **Task Scheduler**. However, what if you're running Windows PowerShell commands or scripts, or Windows tools that don't write their output to a file? If output is important to you, a better choice is using a Windows PowerShell scheduled job. After that job is in the **Task Scheduler**, you can manipulate it further. You can start or stop it in the **Task Scheduler**. If you want to create multiple scheduled jobs or tasks locally, or even on remote computers, 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. PowerShell Gallery contains thousands of scripts that you can use or modify for your specific use, and these scripts are separated into various categories.

For example, there are hundreds of viable scripts that you can run against Active Directory Domain Services (AD DS) and other Active Directory services. Some of these scripts can be very useful. An example is the script that finds user accounts that haven't been used for more than 90 days and then disables them. This can help 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**. Schedule it to run every week and provide a report about what accounts, if any, were disabled.

Demonstration: Create two types of jobs

In this demonstration, you will see how to create and manage two types of jobs.

The demonstration steps should be carried out on the **LON-CL1** VM in the Windows PowerShell ISE. Be sure to run ISE as Administrator. If the ISE is not open, you should open it now, and then open the file **E:\Mod11\Democode\Background.ps1.txt**.

Demonstration steps

1. On **LON-CL1**, open the Windows PowerShell ISE as Administrator.
2. Type the following command, and then press the Enter key:

   Enable-PSRemoting

3. Type the following command, and then press the Enter key:

```
Start-Job –ScriptBlock { Dir C:\ -Recurse } –Name LocalDir
```

4. Type the following command, and then press the Enter key:

```
Invoke-Command –ScriptBlock { Get-EventLog –LogName Security –Newest 100 } –ComputerName
LON-CL1,LON-DC1 –JobName RemoteLogs
```

5. Type the following command, and then press the Enter key:

```
Get-Job
```

6. Type the following command, and then press the Enter key:

```
Get-Job –Name LocalDir | Stop-Job
```

7. Type the following command, and then press the Enter key:

```
Receive-Job  –Name LocalDir
```

8. Type the following command, and then press the Enter key:

```
Remove-Job –Name LocalDir
```

9. Type the following command, and then press the Enter key:

```
Get-Job
```

Repeat this step until the RemoteLogs job shows a status of Completed.

1. Type the following command, and then press the Enter key:

```
Get-Job –Name RemoteLogs -IncludeChildJob | Where location -eq 'LON-DC1' | Select -ExpandProp-
erty ID
```

   Note: Note the job ID that corresponds to the **LON-DC1** job.

2. Type the following command, replacing `<id>` with the job ID number you noted from the previous step, and then press the Enter key:

```
Get-Job –ID <id> | Receive-Job –Keep
```

3. Type the following command, and then press the Enter key:

```
Receive-Job –Name RemoteLogs
```

4. Type the following command, and then press the Enter key:

```
Remove-Job –Name RemoteLogs
```

5. Leave the Windows PowerShell ISE open for the next demonstration.

What are scheduled jobs?

Scheduled jobs are a useful combination of Windows PowerShell background jobs and Windows **Task Scheduler** tasks. Similar to the latter, scheduled jobs are saved to disk. You can review and manage Windows PowerShell scheduled jobs in the **Task Scheduler**, enabling and disabling tasks or simply running the scheduled job. You can even use the scheduled job:

- As a template for creating other scheduled jobs.
- To establish a one-time schedule or periodic schedule for starting jobs.
- To set conditions under which jobs start again.

Note: You can do all of these tasks from the **Task Scheduler**.

Windows PowerShell saves the results of scheduled jobs to disk and creates a running log of job output. Scheduled jobs have a customized set of commands that you can use to manage them. You can use these commands to create, edit, manage, disable, and re-enable scheduled jobs, job triggers, and job options.

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's in the **Microsoft\Windows\PowerShell\ScheduledJobs** path in the **Task Scheduler**. After you create a scheduled job, review and manage it in the **Task Scheduler** by selecting a scheduled job to:

- Find the job triggers on the **Triggers** tab.
- Find the scheduled job options on the **General** and **Conditions** tabs.
- Review the job instances that have already been run on the **History** tab.

Note: When you change a scheduled job setting in **Task Scheduler**, the change applies for all future instances of that scheduled job.

The commands that work with scheduled jobs in the **PSScheduledJob** module are included in the current versions of the Windows Server and Client operating systems. To review 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 running 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**.

Job options and job triggers

To further configure jobs, you can use parameters for job options and define job triggers.

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, such as:

- –*HideInTaskScheduler*, which prevents the job from displaying in the **Task Scheduler**. If you don't include this option, the final job will display in the **Task Scheduler** graphical user interface (GUI).
- –*RunElevated*, which configures the job to run under elevated permissions.
- –*WakeToRun*, which wakes the computer when the job is scheduled to run.

Use other parameters to configure jobs that run when the computer is idle and other options. Many parameters correspond to options in the **Task Scheduler** GUI.

Create a new option object and store it in a variable by using the following command:

```
$opt = New-ScheduledJobOption –RequireNetwork –RunElevated -WakeToRun
```

You don't need to create an option object if you don't 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 one time only. 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'll typically use –*At* to specify a time and use –*DaysOfWeek* to define the days the job should run.
- –*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's 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 signs in.

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'
```

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. You also could 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 run before the job starts.
- –*MaxResultCount* is optional and specifies the maximum number of result sets to store on disk. After this number is reached, the shell deletes older results to make room for new ones. The default value for the -*MaxResultCount* parameter 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).AddMin-
utes(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 in **\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs.**

You can run **Get-ScheduledJob** to review 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 scheduled job results

Because scheduled jobs can run when Windows PowerShell isn't running, results are stored on disk in XML files. If you create a job by using the –*MaxResultCount* parameter, the shell automatically deletes old XML files to make room for new ones. This deletion ensures that no more XML files exist than were specified in the –*MaxResultCount* parameter.

After a scheduled job finishes, running **Get-Job** in Windows PowerShell displays the scheduled job's results as a job object.

Here's an example:

```
PS C:\> Get-Job

Id      Name       PSJobTypeName    State          HasMoreData      Location
Command
```

```
--     ----      -------------   -----         -----------     --------
-------
6      LocalDir  PSScheduledJob  Completed     True            localhost
Dir C:\
```

You can use **Receive-Job** to get a scheduled job's results. If you don't specify *–Keep*, you can receive a job's results 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----         7/26/2021  12:33 AM            PerfLogs
d-r--        11/28/2021   1:54 PM            Program Files
d-r--        12/28/2021   2:22 PM            Program Files (x86)
d----        11/16/2021   9:33 AM            reports
d----         9/18/2021   7:28 AM            Review
d----          1/5/2022   7:49 AM            scr
d----          1/5/2022   7:50 AM            scrx
d-r--         9/15/2021   8:16 AM            Users
d----        12/19/2021   3:24 AM            Windows
-a---          1/1/2022   9:39 AM    2892628 EventReport.html
-a---          1/2/2022  12:37 PM         82 Get-DiskInfo.ps1
-a---        12/30/2021  12:33 PM        246 test.ps1
```

Each time the scheduled job runs, Windows PowerShell creates a new job object to represent the results of the most recent job that ran. You can use **Remove-Job** to remove a job and delete its results file from disk, as the following example depicts:

```
PS C:\> Get-Job -id 6 | Remove-Job
```

Demonstration: Using scheduled jobs

In this demonstration, you'll learn how to create, run, and retrieve the results from a scheduled job.

Demonstration steps

1. On **LON-CL1**, in Windows PowerShell Integrated Scripting Environment (ISE), enter the following command, and then press the Enter key:

   Get-Job | Remove-Job

Note: You might notice an error stating that the directory name **C:\Users\.....\PowerShell\ScheduledJobs** is invalid. This displays if there are no scheduled jobs defined and is expected. Run the **Remove-Job** command to clear the jobs before proceeding with the next steps.

2. Enter the following command, and then press the Enter key:

```
$trigger = New-JobTrigger –Once –At (Get-Date).AddMinutes(2)
```

3. Enter the following command, and then press the Enter key:

```
Register-ScheduledJob –Trigger $trigger –Name DemoJob –ScriptBlock { Get-EventLog –LogName Application }
```

4. Enter the following command, and then press the Enter key:

```
Get-ScheduledJob | Select –Expand JobTriggers
```

Note: Notice the time.

5. Enter the following command, and then press the Enter key:

```
Get-ScheduledJob
```

6. Enter the following command, and then press the Enter key:

```
Get-Job
```

7. Enter the following command, and then press the Enter key:

```
Receive-Job –Name DemoJob
```

8. Enter the following command, and then press the Enter key:

```
Get-Job –Name DemoJob | Remove-Job
```

Test your knowledge

Use the following question to check what you've learned in this lesson.

Question 1

Why would you use `Register-ScheduledJob` from the `PSScheduledJob` module instead of a command in the `ScheduledTasks` module?

Module 11 lab and review

Lab: Jobs management with PowerShell

Scenario

Background jobs provide a useful way to run multiple commands simultaneously and long-running commands in the background. In this lab, you'll learn to create and manage two of the three basic kinds of jobs.

You'll create and configure two scheduled jobs. You'll also create a scheduled task using a Windows PowerShell script that searches for and removes disabled accounts from a certain security group.

Objectives

After completing this lab, you'll be able to:

- Start and manage jobs.
- Create a scheduled job.

Estimated time: 30 minutes

Module review

Use the following question to check what you've learned in this module.

Question 1

What's the main difference between a background job and a scheduled job?

Answers

Question 1

What are examples of tasks that you might want to run in the background?

Any long-running task is an appropriate candidate for running in the background. Also, remember that background jobs can run in parallel, thereby enabling a script to start several tasks that can run concurrently. The script can start the jobs and wait until they all complete before proceeding.

Question 1

Why would you use \`Register-ScheduledJob\` from the \`PSScheduledJob\` module instead of a command in the \`ScheduledTasks\` module?

The \`ScheduledTasks\` module isn't designed to retrieve job results. It's designed to manage the task objects in the Windows Task Scheduler. The commands in \`PSScheduledJob\` manage a type of job that combines the abilities of the Windows Task Scheduler with Windows PowerShell manageability.

Question 1

What's the main difference between a background job and a scheduled job?

A background job runs only while Windows PowerShell is running. A scheduled job can run even if Windows PowerShell isn't running, and you can still use it to retrieve job results.