

**Microsoft®**

MCT USE ONLY. STUDENT USE PROHIBITED

OFFICIAL MICROSOFT LEARNING PRODUCT

**10961B  
Automating Administration  
with Windows PowerShell®**

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2013 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at

<http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners

Product Number: 10961B

Part Number: X18-84549

Released: 3/2014

## 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.**

- a. "Authorized Learning Center" means a Microsoft IT Academy Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
- b. "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.
- c. "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.
- d. "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 a MPN Member, or (iii) a Microsoft full-time employee.
- e. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
- f. "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.
- g. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals and developers on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics or Microsoft Business Group courseware.
- h. "Microsoft IT Academy Program Member" means an active member of the Microsoft IT Academy Program.
- i. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
- j. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals and developers on Microsoft technologies.
- k. "MPN Member" means an active silver or gold-level Microsoft Partner Network program member in good standing.

- I. "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.
  - m. "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.
  - n. "Trainer" means (i) an academically accredited educator engaged by a Microsoft IT Academy Program Member to teach an Authorized Training Session, and/or (ii) a MCT.
  - o. "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.
- a. **If you are a Microsoft IT Academy Program Member:**
    - i. 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.
    - ii. 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,
    - iii. **provided you comply with the following:**
      - iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      - iv. 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,
      - v. 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,
      - vi. 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,

- vii. 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,
  - viii. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
  - ix. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.
- b. **If you are a Microsoft Learning Competency Member:**
- i. 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.
  - ii. 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 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) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content,
- provided you comply with the following:**
- iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
  - iv. 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,
  - v. 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,
  - vi. 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,
  - vii. 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 your Authorized Training Sessions,
  - viii. 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,
  - ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
  - x. you will only provide access to the Trainer Content to Trainers.

c. **If you are a MPN Member:**

- i. 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.
- ii. 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, **provided you comply with the following:**

- iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
- iv. 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,
- v. 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,
- vi. 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,
- vii. 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,
- viii. 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,
- ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
- x. you will only provide access to the Trainer Content to Trainers.

d. **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.

e. **If you are a Trainer:**

- i. 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.

- ii. 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. 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 Programs and Services.** The Licensed Content may contain third party programs or services. These license terms will apply to your use of those third party programs or services, unless other terms accompany those programs and services.

**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:

- a. **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.
- b. **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 software, 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 software, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.
- c. **Pre-release Term.** If you are an Microsoft IT Academy Program Member, Microsoft Learning Competency Member, MPN Member 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](http://www.microsoft.com/exporting).
7. **SUPPORT SERVICES.** Because the Licensed Content is "as is", we may not 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.**
  - a. 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.

- b. 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 contrefaçon 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 September 2012

# Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.
- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance<sup>1</sup>. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.
- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning  
[www.microsoft.com/learning](http://www.microsoft.com/learning)



<sup>1</sup> IDC, Value of Certification: Team Certification and Organizational Performance, November 2006

## Acknowledgments

Microsoft Learning would like to acknowledge and thank the following for their contribution towards developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

### **Don Jones — Course Designer and Author**

Don is a 10-year recipient of Microsoft's Most Valuable Professional (MVP) Award, the author of more than 50 IT books, and a co-founder of PowerShell.org, Inc. He writes the Windows PowerShell column for Microsoft TechNet Magazine, and is one of the world's most well-known independent experts on Windows PowerShell.

### **Jeffery Hicks — Technical Reviewer**

Jeff is a multi-year PowerShell MVP and Microsoft Certified Trainer. He writes the Prof. PowerShell column for MCPMag.com, has authored a number of books on scripting and Windows PowerShell, and has trained hundreds of IT Pros around the world.

### **Jason Yoder — Technical Reviewer**

Jason has been an MCP for 15 years and an MCT for the last 5. Currently he is president of MCTExpert Inc., and works as an IT trainer delivering Microsoft courses on Windows PowerShell and Windows Server to IT professionals from Maine to Hawaii. Jason has designed and managed Windows environments in the military, public, and private sectors on all of Microsoft's platforms since Windows NT 4. He is also a contributing moderator to the Active Directory forum on PowerShell.com and has published over 700 blog posts read by over a half million IT professionals.

### **Aleksander Nikolic — Technical Reviewer**

Aleksandar is a Microsoft Most Valuable Professional (MVP) for Windows PowerShell, a co-founder of PowerShellMagazine.com, and the community manager of PowerShell.com. He is an experienced presenter and speaker about Microsoft automation solutions, and has more than 17 years of experience as a system administrator.

### **Dr. Tobias Weltner — Technical Reviewer**

Tobias is a long-term Microsoft's MVP Award recipient and has written more than 130 IT books with Microsoft Press and others. As a PowerShell pioneer of the early days, he now operates the PowerShell online magazine [www.powertheshell.com](http://www.powertheshell.com), organizes the European PowerShell conference, and is one of Europe's leading PowerShell trainers and experts.

# Contents

## Module 1: Getting Started with Windows PowerShell

Module Overview	01-1
<b>Lesson 1:</b> Overview and Background	01-2
<b>Lab A:</b> Configuring Windows PowerShell	01-10
<b>Lesson 2:</b> Finding and Learning Commands	01-12
<b>Lesson 3:</b> Running Commands	01-20
<b>Lab B:</b> Finding and Running Basic Commands	01-26
Module Review and Takeaways	01-29

## Module 2: Working with the Pipeline

Module Overview	02-1
<b>Lesson 1:</b> Understanding the Pipeline	02-2
<b>Lesson 2:</b> Selecting, Sorting, and Measuring Objects	02-6
<b>Lab A:</b> Using the Pipeline	02-13
<b>Lesson 3:</b> Converting, Exporting, and Importing Objects	02-16
<b>Lab B:</b> Converting, Exporting, and Importing Objects	02-21
<b>Lesson 4:</b> Filtering Objects Out of the Pipeline	02-24
<b>Lab C:</b> Filtering Objects	02-30
<b>Lesson 5:</b> Enumerating Objects in the Pipeline	02-33
<b>Lab D:</b> Enumerating Objects	02-36
Module Review and Takeaways	02-38

## Module 3: Understanding How the Pipeline Works

Module Overview	03-1
<b>Lesson 1:</b> Passing Data in the Pipeline By Value	03-2
<b>Lesson 2:</b> Passing Data in the Pipeline By Property Name	03-7
<b>Lab:</b> Working with Pipeline Parameter Binding	03-11
Module Review and Takeaways	03-14

**Module 4: Using PSProviders and PSDrives**

Module Overview	<b>04-1</b>
<b>Lesson 1:</b> Using PSProviders	<b>04-2</b>
<b>Lesson 2:</b> Using PSDrives	<b>04-5</b>
<b>Lab:</b> Using PSProviders and PSDrives	<b>04-9</b>
Module Review and Takeaways	<b>04-12</b>

**Module 5: Formatting Output**

Module Overview	<b>05-1</b>
<b>Lesson 1:</b> Using Basic Formatting	<b>05-2</b>
<b>Lesson 2:</b> Using Advanced Formatting	<b>05-5</b>
<b>Lesson 3:</b> Redirecting Formatted Output	<b>05-8</b>
<b>Lab:</b> Formatting Output	<b>05-11</b>
Module Review and Takeaways	<b>05-15</b>

**Module 6: Querying Management Information by Using WMI and CIM**

Module Overview	<b>06-1</b>
<b>Lesson 1:</b> Understanding WMI and CIM	<b>06-2</b>
<b>Lesson 2:</b> Querying Data by Using WMI and CIM	<b>06-6</b>
<b>Lesson 3:</b> Making Changes by Using WMI and CIM	<b>06-13</b>
<b>Lab:</b> Working with WMI and CIM	<b>06-17</b>
Module Review and Takeaways	<b>06-21</b>

**Module 7: Preparing for Scripting**

Module Overview	<b>07-1</b>
<b>Lesson 1:</b> Using Variables	<b>07-2</b>
<b>Lesson 2:</b> Scripting Security	<b>07-9</b>
<b>Lab:</b> Working with Security in Windows PowerShell	<b>07-14</b>
Module Review and Takeaways	<b>07-16</b>

**Module 8: Moving from Command to Script to Module**

Module Overview	<b>08-1</b>
-----------------	-------------

<b>Lesson 1:</b> Moving from Command to Script	<b>08-2</b>
<b>Lab A:</b> Moving from Command to Script	<b>08-7</b>
<b>Lesson 2:</b> Moving from Script to Function to Module	<b>08-11</b>
<b>Lab B:</b> Moving from Script to Function to Module	<b>08-16</b>
<b>Lesson 3:</b> Implementing Basic Error Handling	<b>08-20</b>
<b>Lab C:</b> Implementing Basic Error Handling	<b>08-24</b>
<b>Lesson 4:</b> Using Basic Scripting Constructs	<b>08-26</b>
<b>Lab D:</b> Creating an Advanced Function	<b>08-30</b>
<b>Lesson 5:</b> Exploring Other Scripting Features	<b>08-34</b>
Module Review and Takeaways	<b>08-36</b>
<b>Module 9: Administering Remote Computers</b>	
Module Overview	<b>09-1</b>
<b>Lesson 1:</b> Using Basic Remoting	<b>09-2</b>
<b>Lesson 2:</b> Using Advanced Remoting Techniques	<b>09-12</b>
<b>Lab A:</b> Using Basic Remoting	<b>09-15</b>
<b>Lesson 3:</b> Using Remoting Sessions	<b>09-18</b>
<b>Lab B:</b> Using Remoting Sessions	<b>09-23</b>
Module Review and Takeaways	<b>09-26</b>
<b>Module 10: Putting it All Together</b>	
Module Overview	<b>10-1</b>
<b>Lesson 1:</b> Provisioning a New Server Core Instance	<b>10-2</b>
<b>Lab:</b> Provisioning a New Server Core Instance	<b>10-6</b>
Module Review and Takeaways	<b>10-14</b>
<b>Module 11: Using Background Jobs and Scheduled Jobs</b>	
Module Overview	<b>11-1</b>
<b>Lesson 1:</b> Using Background Jobs	<b>11-2</b>
<b>Lab A:</b> Using Background Jobs	<b>11-8</b>
<b>Lesson 2:</b> Using Scheduled Jobs	<b>11-11</b>
<b>Lab B:</b> Using Scheduled Jobs	<b>11-16</b>

Module Review and Takeaways	<b>11-19</b>
-----------------------------	--------------

**Module 12: Using Profiles and Advanced Windows PowerShell Techniques**

Module Overview	<b>12-1</b>
<b>Lesson 1:</b> Using Advanced Windows PowerShell Techniques	<b>12-2</b>
<b>Lesson 2:</b> Creating Profile Scripts	<b>12-11</b>
<b>Lesson 3:</b> Working with Alternative Credentials	<b>12-15</b>
<b>Lab:</b> Practicing Advanced Techniques	<b>12-18</b>
Module Review and Takeaways	<b>12-22</b>

**Lab Answer Keys**

<b>Module 1 Lab A:</b> Configuring Windows PowerShell	<b>L01-1</b>
<b>Module 1 Lab B:</b> Finding and Running Basic Commands	<b>L01-3</b>
<b>Module 2 Lab A:</b> Using the Pipeline	<b>L02-1</b>
<b>Module 2 Lab B:</b> Converting, Exporting, and Importing Objects	<b>L02-3</b>
<b>Module 2 Lab C:</b> Filtering Objects	<b>L02-6</b>
<b>Module 2 Lab D:</b> Enumerating Objects	<b>L02-9</b>
<b>Module 3 Lab:</b> Working with Pipeline Parameter Binding	<b>L03-1</b>
<b>Module 4 Lab:</b> Using PSProviders and PSDrives	<b>L04-1</b>
<b>Module 5 Lab:</b> Formatting Output	<b>L05-1</b>
<b>Module 6 Lab:</b> Working with WMI and CIM	<b>L06-1</b>
<b>Module 7 Lab:</b> Working with Security in Windows PowerShell	<b>L07-1</b>
<b>Module 8 Lab A:</b> Moving from Command to Script	<b>L08-1</b>
<b>Module 8 Lab B:</b> Moving from Script to Function to Module	<b>L08-6</b>
<b>Module 8 Lab C:</b> Implementing Basic Error Handling	<b>L08-11</b>
<b>Module 8 Lab D:</b> Creating an Advanced Function	<b>L08-13</b>
<b>Module 9 Lab A:</b> Using Basic Remoting	<b>L09-1</b>
<b>Module 9 Lab B:</b> Using Remoting Sessions	<b>L09-4</b>
<b>Module 10 Lab:</b> Provisioning a New Server Core Instance	<b>L10-1</b>
<b>Module 11 Lab A:</b> Using Background Jobs	<b>L11-1</b>
<b>Module 11 Lab B:</b> Using Scheduled Jobs	<b>L11-3</b>
<b>Module 12 Lab:</b> Practicing Advanced Techniques	<b>L12-1</b>

# About This Course

This section provides you with a brief description of the course—*10961B: Automating Administration with Windows PowerShell* — audience, suggested prerequisites, and course objectives.

## Course Description

This five day course is designed to teach IT professionals the core skills needed to use Windows PowerShell 4.0 to automate administrative tasks. It is mostly also applicable to previous versions such as version 2.0 and 3.0. Many IT professionals may use a mixed environment that can include Windows PowerShell v2.0, v3.0, and v4.0. Dependencies of some server products, such as Microsoft Exchange Server 2010 depending upon Windows PowerShell 2.0, often drive this mixed-version requirement, so where there are specific differences that is called out. The course uses Microsoft Windows 8.1 and Microsoft Windows Server 2012 R2 for examples and labs, but the skills taught in this course are applicable to Microsoft Exchange Server 2010 and later, Microsoft SharePoint Server 2010 and later, Microsoft SQL Server 2008 R2 and later, Microsoft Windows Server 2008 R2, Microsoft Windows 7, and other products that use Windows PowerShell. This course does not focus primarily on scripting or programming, although it does include lessons that feature basic scripting tasks. The course focuses mainly on using Windows PowerShell as an interactive command line interface. Major Windows PowerShell feature coverage includes remoting, background jobs, scheduled jobs, the pipeline, Windows Management Instrumentation (WMI) and Common Information Model (CIM), output formatting, output conversion, and exporting.

## Audience

This course is intended for students who want to use Windows PowerShell to automate administrative tasks from the command line, using any Microsoft or independent software vendor (ISV) product that supports Windows PowerShell manageability. This course is not intended to be a scripting or programming course, and includes only basic coverage of scripting and programming topics. Students are not expected to have prior scripting or programming experience, and are not expected to have prior Windows PowerShell experience.

## Student Prerequisites

This course requires that you have the ability to meet following prerequisites:

- Previous Windows Server and Windows Client management knowledge and hands on experience.
- Experience Installing and Configuring Windows Server into existing enterprise environments, or as standalone installations.
- Knowledge and experience of network adapter configuration, basic Active Directory user administration, and basic disk configuration.
- Knowledge and hands on experience specifically with Windows Server 2012 or Windows Server 2012 R2 and Windows 8 or Windows 8.1 would be of benefit but is not essential.
- Students who have prior experience in a scripting or programming language may have an easier time with some of this course's advanced concepts but previous scripting or programming experience is not required.

## Course Objectives

After completing this course, students will be able to:

- Understand the basic concepts behind Windows PowerShell.
- Work with the Pipeline.
- Understand How the Pipeline Works.
- Use PSProviders and PSDrives.
- Format Output.
- Use WMI and CIM.
- Prepare for Scripting.
- Moving From a Command to a Script to a Module.
- Administer Remote Computers.
- Put the various Windows PowerShell components together.
- Use Background Jobs and Scheduled Jobs.
- Use Advanced PowerShell Techniques and Profiles.

## Course Outline

The course outline is as follows:

### **Module 1, "Getting Started with Windows PowerShell"**

This module introduces students to Windows PowerShell, explains the two built-in host applications, and teaches students to configure the host applications.

### **Module 2, "Working with the Pipeline"**

This module covers core Windows PowerShell techniques and commands, including customizing command output, exporting and converting data, sorting objects, filtering objects, and enumerating objects.

### **Module 3, "Understanding How the Pipeline Works"**

This module explains how Windows PowerShell passes objects from command to common within the pipeline. Students are given several examples and learn to explain the pipeline operation and predict command behavior.

### **Module 4, "Using PSProviders and PSDrives"**

This module explains the purpose and use of Windows PowerShell providers and drives, and shows students how to use these components for administrative tasks.

### **Module 5, "Formatting Output"**

This module demonstrates how to format command output and how to create custom output elements.

### **Module 6, "Using WMI and CIM"**

This module explains Windows Management Instrumentation (WMI) and Common Information Model (CIM), and shows students how to use these technologies.

### **Module 7, "Preparing for Scripting"**

This module prepares students for scripting by explaining Windows PowerShell's security model and formally covering variables.

### **Module 8, "Moving From a Command to a Script to a Module"**

This module shows students how to take a command and turn it into a parameterized script, and how to evolve that script into a standalone script module. Students therefore learn the foundations needed to create their own reusable tools.

**Module 9**, "Administering Remote Computers"

This module explains Windows PowerShell remoting, and shows students how to use remoting to manage multiple remote computers.

**Module 10**, "Putting it All Together"

This module offers students an opportunity to use everything they have learned so far. Students will discover, learn, and run commands that perform a complex, real-world administrative task.

**Module 11**, "Using Background Jobs and Scheduled Jobs"

In this module students will learn to create and manage background jobs and scheduled jobs.

**Module 12**, "Using Advanced PowerShell Techniques and Profiles"

This module includes a variety of advanced techniques, including additional comparison operators, using alternate credentials, creating profile scripts, manipulating string and date objects, and so on.

## Exam/Course Mapping

This course, *10961B: Automating Administration with Windows PowerShell*, does not have a direct mapping to any exam.

However, while there is not a standalone Windows PowerShell exam it is covered across all the individual Microsoft Certified Solutions Associate (MCSA) and Microsoft Certified Certified Solutions Expert (MCSE) exams and this course will help prepare you for Windows PowerShell related concepts and processes within those exams.

## Course Materials

The following materials are included with your kit:

- **Course Handbook** A succinct classroom learning guide that provides all the critical technical information in a crisp, tightly-focused format, which is just right for an effective in-class learning experience.

You may be accessing either a printed course hand book or digital courseware material via the Arvato Skillpipe reader. Your Microsoft Certified Trainer will provide specific details but both contain the following:

- **Lessons:** Guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
- **Labs:** Provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
- **Module Reviews and Takeaways:** Provide improved on-the-job reference material to boost knowledge and skills retention.
- **Lab Answer Keys:** Provide step-by-step lab solution guidance at your fingertips when it's needed.

**Course Companion Content on the <http://www.microsoft.com/learning/companionmoc> site:**

Searchable, easy-to-navigate digital content with integrated premium on-line resources designed to supplement the Course Handbook.

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

**Student Course files on the <http://www.microsoft.com/learning/companionmoc> site:** Includes

the Allfiles.exe, a self-extracting executable file that contains all the files required for the labs and demonstrations.

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

## Virtual Machine Environment

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

### Virtual Machine Configuration

In this course, you will use Hyper-V® in a Windows Server 2012 host to perform the labs.



**Important** If you are working in a local lab environment at the end of each lab, you must revert the virtual machine to a snapshot and not save any changes. Steps on how to do this are included at the end of each lab.

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

Virtual machine	Role
10961B-LON-DC1	Windows Server 2012 R2 Server Core domain controller and DHCP server for the Adatum.com domain.
10961B-LON-CL1	Windows 8.1 client computer and in the Adatum.com domain with Remote Server Administration Tools (RSAT) installed.
10961B-LON-SVR1	Newly installed Windows Server 2012 R2 Server Core computer.

### Software Configuration

The following software is installed on each virtual machine:

- Remote Server Administration Toolkit (RSAT) is installed on LON-CL1.

## Course Files

There are lab files associated with the labs in this course. The lab files are located in the folder E:\Allfiles\ModXX\Labfiles on LON-CL1.

## Classroom Setup

Each classroom computer will have the same virtual machines configured in the same way.

You may be accessing the lab virtual machines in either in a hosted online environment with a web browser or by using Hyper-V on a local machine. The labs and virtual machines are the same in both scenarios however there may be some slight variations because of hosting requirements. Any discrepancies will be called out in the Lab Notes on the hosted lab platform.

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

## Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware are taught.

- Hardware level 6 with 8 gigabytes (GB) of random access memory (RAM)

## Navigation in Windows Server 2012

If you are not familiar with the user interface in Windows Server 2012 or Windows 8 then the following information will help orient you to the new interface.

- Sign in and Sign out replace Log in and Log out.
- Administrative tools are found in the Tools menu of Server Manager.
- Move your mouse to the lower right corner of the desktop to open a menu with:
  - Settings: This includes Control Panel and Power
  - Start menu: This provides access to some applications
  - Search: This allows you to search applications, settings, and files

You may also find the following shortcut keys useful:

- Windows: Opens the Start menu
- Windows+C: Opens the same menu as moving the mouse to the lower right corner
- Windows+I: Opens Settings
- Windows+R: Opens the Run window

## Additional Learning Resources

Additional training and content resources are also available on the Microsoft Virtual Academy site, <http://www.MicrosoftVirtualAcademy.com>, where you can view further study resources and online courses related to this and other technologies. There are a variety of resource and formats available on this site to assist you with general skills/career development and specific exam preparation.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 1

## Getting Started with Windows PowerShell

### Contents:

Module Overview	01-1
<b>Lesson 1:</b> Overview and Background	01-2
<b>Lab A:</b> Configuring Windows PowerShell	01-10
<b>Lesson 2:</b> Finding and Learning Commands	01-12
<b>Lesson 3:</b> Running Commands	01-20
<b>Lab B:</b> Finding and Running Basic Commands	01-26
Module Review and Takeaways	01-29

## Module Overview

Even if you are familiar with command-line interfaces from other operating systems, or if you have used the Cmd.exe shell in Windows, you must learn (and in some cases, re-learn) important concepts and techniques before you can start to be effective with Windows PowerShell.

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

 **Additional Reading:** You may be interested in the Windows PowerShell team blog, located at <http://go.microsoft.com/fwlink/?LinkId=306142>.

 **Additional Reading:** If you need community-based help for Windows PowerShell, visit [PowerShell.org](http://PowerShell.org) or the Microsoft Developer Network Scripting Forum at <http://go.microsoft.com/fwlink/?LinkId=306143>.

### Objectives

After completing this module, students will be able to:

- Open and configure Windows PowerShell
- Discover, learn, and run Windows PowerShell commands
- Run commands by using correct command and parameter syntax

## Lesson 1

# Overview and Background

It is easy to overlook the background and intended purpose of Windows PowerShell and to just jump in and start using it. But understanding where Windows PowerShell comes from, and what it is intended to do, can help you use it more easily and more effectively.

In this lesson, you will learn about Windows PowerShell's system requirements and learn to open and configure the two included host applications.

### Lesson Objectives

After completing this lesson, students will be able to:

- Describe the purpose of Windows PowerShell
- List the major versions of Windows PowerShell
- Describe the difference between shell features and operating system features
- Identify the two native Windows PowerShell hosting applications
- Display Windows PowerShell version information
- Describe common mistakes made when you open the shell
- Configure the Windows PowerShell console host
- Configure the Windows PowerShell ISE host

### Windows PowerShell Overview

Introduced in 2006, Windows PowerShell is an object-oriented engine for administrative automation. It is not technically a command-line interface (CLI), although a CLI is the primary means through which you will interact with the shell. Windows PowerShell can also be hosted within other applications. This includes graphical user interface (GUI) applications that use Windows PowerShell behind the scenes to do tasks.

This architecture and the ability to use Windows PowerShell directly as a CLI, or to use it through a GUI that embeds the shell, is intended to help increase consistency and coverage for administrative capabilities. For example, an administrator might rely completely upon a GUI application to perform tasks. However, if the administrator must perform some task or implement some process that the GUI does not explicitly support, the administrator can use the shell directly instead. When correctly implemented, this architecture helps to ensure that anything that you can do in the GUI can also be done in the CLI, with the CLI offering the additional ability to customize processes and procedures.

- Introduced in 2006
- Implemented as an engine that can be embedded in a GUI or used directly as a CLI
- Functionality provided by commands:
  - Cmdlets (pronounced "command-lets")
  - Functions
  - Workflows
  - More

Windows PowerShell's main functionality is provided by *commands*. These come in many varieties: cmdlets (pronounced "command-lets"), functions, workflows, and more. These commands are building blocks, designed to be pieced together to implement complex and customized processes and procedures.

## Windows PowerShell Versions

Windows PowerShell 1.0 was the initial release, and it was available as an installation option in Windows Vista and Windows Server 2008.

Currently, no applications or server products have a firm dependency on Windows PowerShell 1.0, and most computers should be fine having at least Windows PowerShell 2.0 installed. Windows PowerShell 1.0 was compatible with Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008.

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

Windows PowerShell 3.0 is installed by default on Windows 8 and Windows Server 2012, and is available as an out-of-band free download for Windows 7 with Service Pack 1 (SP1), Windows Server 2008 (SP2), and Windows Server 2008 R2 (SP1). Be aware that Windows PowerShell 3.0 is incompatible with Windows XP, Windows Server 2003, and Windows Vista. The out-of-band free download package in which Windows PowerShell 3.0 is available is the Windows Management Framework 3.0. As well as Windows PowerShell v3.0 it also includes Windows Management Instrumentation (WMI), WinRM, Management Open Data Protocol (OData) Internet Information Services (IIS) Extension, and Server Manager Common Information Model (CIM) Provider updates. The WinRM feature in Windows PowerShell 3.0 is compatible with the same feature in Windows PowerShell 2.0. Therefore, the two versions can successfully communicate with one another for management.

Windows PowerShell 3.0 requires version 4.0 of the Microsoft® .NET Framework; older versions of Windows PowerShell required Windows PowerShell 2.0 of the .NET Framework.

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

Be aware that many server products have a dependency on a specific version of Windows PowerShell. For example, Microsoft Exchange Server 2010 requires Windows PowerShell 2.0, and is not compatible with later versions of Windows PowerShell. Before installing a new version of Windows PowerShell on a computer, verify that version's compatibility with the software installed on that computer.



**Reference Links:** The latest Windows Management Framework version can be found in the Microsoft Download Center.

	2.0	3.0	4.0
<b>Windows XP</b>	Available	No	No
<b>Windows Server 2003</b>	Available	No	No
<b>Windows Vista</b>	Available	No	No
<b>Windows Server 2008</b>	Available	Available with SP2	No
<b>Windows 7</b>	Installed	Available with SP1	Available
<b>Windows Server 2008 R2</b>	Installed	Available with SP2	Available
<b>Windows 8</b>	No	Installed	Available
<b>Windows Server 2012</b>	No	Installed	Available
<b>Windows 8.1 and Windows Server 2012 R2</b>	No	No	Installed
<b>Windows PowerShell 1.0 and 2.0 require .NET Framework 2.0</b>			
<b>Windows PowerShell 3.0 requires .NET Framework 4.0</b>			
<b>Windows PowerShell 4.0 requires .NET Framework 4.5</b>			

## Windows PowerShell vs. Operating System

Each version of Windows PowerShell includes certain core, native functionality. This functionality is available regardless of which operating system version Windows PowerShell runs on. For example, Windows PowerShell 3.0 introduces a new `Invoke-WebRequest` command, and that command is available whether Windows PowerShell is running on Windows 7, Windows 8, or on a server operating system.

However, much of Windows PowerShell's usefulness stems from its extensibility, and extensions are not provided with the shell itself.

For example, Windows 8 and Windows Server 2012 include new commands that enable network management, scheduled task management, and more. Although these commands *require* Windows PowerShell 3.0, they are not *included* in Windows PowerShell 3.0. That is, were you to install Windows PowerShell 3.0 on Windows 7, you would not gain these additional commands, because they are technically features of Windows 8 and Windows Server 2012 and not features of Windows PowerShell itself.

In other words, one advantage of moving to the newer operating system versions is that they provide more support for management based on Windows PowerShell.

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

## Working in Mixed-Version Environments

Many organizations will have some computers running Windows PowerShell 2.0, others running Windows PowerShell 3.0, and still others running Windows PowerShell 4.0 or later. Sometimes, these mixed-version environments are the result of organization policies that do not permit installing newer versions of Windows PowerShell. Sometimes, the mixed-version environment is the result of software products, especially server software, that have a dependency on a particular version of Windows PowerShell.

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

- Many organizations must operate environments that contain more than one version of Windows PowerShell.
- This course attempts to make version-specific differences as clear as possible, to support professionals that must work in a mixed-version environment.

## Two Host Applications

To interact with Windows PowerShell, you must use an application that embeds, or *hosts*, the shell's engine. Some of those applications will be graphical user interfaces, such as the Exchange Management Console (EMC) in Microsoft Exchange Server 2007 and later versions. In this course, you will primarily interact with Windows PowerShell through one of the two command-line interface (CLI) hosts provided by Microsoft, that is, the console and the Integrated Scripting Environment, or ISE.

- Console
  - Basic command-line interface
  - Maximum support for PowerShell features
  - Minimal editing capabilities
- ISE
  - Script editor and console combination
  - Some PowerShell features not supported
  - Rich editing capabilities
- Third-party hosting applications/editors
  - Varying features and pricing

The console uses the Windows built-in console host application. This is basically the same command-line experience offered by the older Cmd.exe shell. It provides a straightforward but simple environment, does not support double-byte character sets, and has limited editing capabilities. It currently provides the broadest Windows PowerShell functionality. This includes the ability to capture a text transcript of your shell activities to a file.

The ISE is a Windows Presentation Foundation (WPF) application that provides rich editing capabilities, IntelliSense code hinting and completion, and support for double-byte character sets. However, it does not support all Windows PowerShell functionality. Most notably, it cannot capture shell activity to a text transcript file. The ISE also does not support multiline commands. For example, in the console, you could type this:

```
PS C:\> Get-Service |  
>> Out-File C:\services.txt  
>>
```

In that example, you press Enter after typing the pipe character (|), and the console allows you to continue typing the command line on the next physical line. Pressing Enter on a blank line executes everything that you typed up until that point. The ISE does not support that technique.

In this course, you will begin by using the console. This is a good way to reinforce important foundational skills. Toward the end of the course, you will shift to using the ISE as you begin to link multiple commands together into scripts. Your instructor may use the ISE in demonstrations, as the ISE makes it easier to run the demonstration scripts supplied with this course.

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

## What Version Are You Running?

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

 **Note:** The language version used in Windows PowerShell 1.0 is also used in Windows PowerShell 1.0, Windows PowerShell 2.0, Windows PowerShell 3.0, and Windows PowerShell 4.0. The language version indicates the keywords used for Windows PowerShell scripts.

- Use **\$PSVersionTable** to determine the version of PowerShell, and its supporting components, that you are using
- In PowerShell 1.0, this will return a blank result
- Run **powershell –version 2.0** to start Windows PowerShell with the PowerShell 2.0 engine
  - Provides backward compatibility
  - Available only on systems where PowerShell 2.0 was installed prior to PowerShell 3.0 or 4.0 being installed

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

When you install Windows PowerShell 3.0 or later on a system that already has Windows PowerShell 2.0 installed, the new version of Windows PowerShell will install side by side, leaving the Windows PowerShell 2.0 engine available for execution. This behavior is meant to help with potential compatibility issues by providing the Windows PowerShell 2.0 engine for applications that require it and that are incompatible with the Windows PowerShell 3.0 (or 4.0) engine. Be aware that running Windows PowerShell in 2.0 mode does not provide 100 percent Windows PowerShell 2.0 compatibility, because some commands' behavior has changed in later versions. Running in 2.0 mode changes only the version of the Windows PowerShell engine used to run those commands.

To run the shell in Windows PowerShell 2.0 mode, in the Windows PowerShell console, run the command **PowerShell.exe –version 2.0** and press Return. If you then run the **\$PSVersionTable** command, the command will display a **PSVersion** value of 2.0.

## Precautions When Opening the Shell

On 64-bit operating systems, Windows PowerShell's host applications are available in both 64-bit (x64) and 32-bit (x86) versions. Typically, you will use the 64-bit version that is shown as **Windows PowerShell** or **Windows PowerShell ISE**. The 32-bit versions are provided for compatibility with locally installed 32-bit shell extensions and are listed as **Windows PowerShell (x86)** or **Windows PowerShell ISE (x86)**. As soon as it is opened, the applications' window title bars reflect the same names. Be certain you are opening the appropriate version.

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

On 32-bit operating systems, Windows PowerShell's host applications are available only in 32-bit versions, and the icons and window title bars do not carry the (x86) designation.

On computers that have User Account Control (UAC) enabled, Windows PowerShell can be opened without Administrative Credentials. Because you will frequently be using the shell to perform administrative tasks, you may have to make sure that the shell opens with full Administrative Credentials. To do this, right-click the application icon and select **Run as Administrator** from the shortcut menu. When you are running with Administrative Credentials, the application's window title bar will say **Administrator**. Make sure that you check this when you open the shell.

## Configuring the Console

The default font that is used by the console application can make it difficult to differentiate between important characters when you use Windows PowerShell, so it is usually a good idea to configure a different font. You may also want to change the font size and colors to make the text easier to read.

When you open the shell, type the following characters and make sure that you can easily differentiate between them:

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

- Select a font that enables easy differentiation between often-confused characters:  
` ' { [ <
- Modify window layout to fit the entire window on-screen and remove the horizontal scroll bar
- Select an alternate color combination for primary text, if desired

If any of these characters seem too similar to one another, try configuring the console to use a different font. TrueType fonts in particular may be easier to read than the default Raster font. To configure a new font, click the control box in the upper-left corner of the console window. From the shortcut menu, select **Properties**, and then select the **Font** tab. You can also select a font size on this tab.

You should also size the shell window to fit completely on your screen so that no command output will be positioned off-screen. In addition, configure the console window not to have a horizontal scroll bar. To do this, open the console property dialog box again, and then select the **Layout** tab. Make sure that both the **Screen Buffer Size** and **Window Size** have the same value that is shown for **Width** (it is fine for the **Height** to be significantly larger for the **Screen Buffer Size**). Close the dialog box and verify that the window fits on the screen and that no horizontal scroll bar is shown. A vertical scroll bar will still be present in the console.

Finally, if you are dissatisfied with the default white-on-blue color scheme, the console's **Properties** dialog box lets you select from a small range of alternative colors. Use the **Colors** tab to do this, noting that you can change only the primary text color. The color of error messages and other output cannot be changed from this dialog box.

Remember that the console host does support Clipboard copying and pasting, although it does not use standard keyboard shortcuts. Use the mouse pointer to highlight a block of text, and press **Enter** to copy

that text to the Clipboard. Right-click the mouse button to paste. For this copy and paste functionality to work, you need to ensure that **QuickEdit Mode** is enabled in the console's **Properties** dialog box, on the Options tab that is in the Edit Options section. On some computers, this mode may be enabled by default.

## Demonstration: Configuring the Console

In this demonstration, you will see how to open the 64-bit console as Administrator, and how to configure the font and layout properties of the console host and start a transcript.

### Demonstration Steps

1. Open the 64-bit Windows PowerShell console host 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 whole window fits on the screen and does not display a horizontal scroll bar.
5. Start a shell transcript.

## Configuring the ISE

The ISE is a fully graphical environment that provides a script editor, debugging capabilities, an interactive console, and several tools that are designed to help you discover and learn new Windows PowerShell commands. Although many of the ISE's capabilities will be covered and demonstrated more fully in later modules, for now you will want to gain a basic familiarity with how it works.

The ISE offers two main panes: a Script Pane, or script editor; and the Console pane. These can be positioned one above the other in a two-pane view, or you can maximize one and switch back and forth between it and the other. By default, a dockable Command Pane is also displayed, which enables you to search for commands, browse available commands, and fill in parameters for a command you select.

The ISE gives you the ability to customize its view in a number of ways. A slider in the lower-right area of the window changes the active font size, whereas the **Options** dialog box lets you customize font and color selection for many different Windows PowerShell text elements, such as keywords, string values, and more. The ISE supports visual themes. It provides several built-in themes and gives you the option to create custom themes. A *theme* is a collection of font and color settings that can be applied as a group to customize the tool's appearance.

Other ISE features include a built-in, extensible *snippets* library; the ability to load functionality add-ins created by Microsoft or by third parties; integration with Windows PowerShell's debugging capabilities; and more.

- Two panes: Script and console
- One-pane and two-pane view options
- Command Pane can be docked or floated
- Customization of font selection, size, and colors
- Bundling of color selections into themes
- Additional features include snippets, add-ins, and debugging, and more

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Configuring the ISE

In this demonstration, you will see how to open the ISE, arrange its user interface elements, and customize its appearance.

### Demonstration Steps

1. Use the Windows PowerShell taskbar icon to open the Windows PowerShell ISE.
2. Open the ISE by running a command from the Windows PowerShell console application.
3. Use toolbar buttons to arrange the Script Pane and Console pane.
4. Dock, undock, and close the Command Pane.
5. Change the font size.
6. Select a color theme.

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

# Lab A: Configuring Windows PowerShell

## Scenario

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

## Objectives

After completing this lab, students will be able to:

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

## Lab Setup

Estimated Time: 15 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

Be aware that the changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - a. User name: **Administrator**
  - b. Password: **Pa\$\$w0rd**
  - c. Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Configure the Windows PowerShell Console Application

### Scenario

In this exercise, you will open the Windows PowerShell console application and configure its appearance and layout.

The main tasks for this exercise are as follows:

1. Start the 64-bit console application as Administrator and pin Windows PowerShell icon to the taskbar
2. Configure the Windows PowerShell console application

### 3. Start a shell transcript

► Task 1: Start the 64-bit console application as Administrator and pin Windows PowerShell icon to the taskbar

1. On the 10961B-LON-CL1 virtual machine, log on as **Adatum\Administrator**.
2. Start the 64-bit console application as Administrator.
3. Pin the **Windows PowerShell** icon to the taskbar.

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

1. Configure the shell to use the Consolas font.
2. Select alternate display colors for primary text.
3. Size the window to fit on the screen and to remove any horizontal scroll bar.

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

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

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

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

## Exercise 2: Configure the Windows PowerShell ISE Application

### Scenario

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

The main tasks for this exercise are as follows:

1. Open the 64-bit Windows PowerShell ISE application as Administrator
2. Customize the appearance of the ISE to use the single-pane view, hide the Command Pane, and adjust the font size

► **Task 1: Open the 64-bit Windows PowerShell ISE application as Administrator**

1. Ensure you are still logged onto the 10961B-LON-CL1 virtual machine as Adatum\Administrator.
2. Open the 64-bit Windows PowerShell ISE application as Administrator.

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

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

**Results:** After completing this lab, you will have customized the appearance of the Windows PowerShell ISE application.

**Question:** Why might you decide to use the console application instead of the ISE?

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

## Lesson 2

# Finding and Learning Commands

In this lesson, you will learn to use Windows PowerShell's Help system to discover new commands, and to learn how to use them. You will also learn the correct syntax for running Windows PowerShell commands, and learn about shortened syntax forms that can be used to lessen how much typing you have to do in the shell.

## Lesson Objectives

After completing this lesson, students will be able to:

- Execute familiar commands that are now implemented as aliases
- Describe the purpose of the shell Help system
- Display Help for a command
- Discover commands by using keyword searches
- Interpret Help file contents
- Update local Help content
- Display "About" file content

## Familiar-Seeming Commands

When you first start using Windows PowerShell, especially for file and folder management, the commands can seem to be familiar. **Dir** gives you a listing of files and folders (as does **Ls**), **Cd** changes folders, and **Type** (or **Cat**) displays the contents of a text file. Run **Dir > directory.txt** to redirect a directory listing into a text file, or run **Mkdir** to create a new directory. Even commands like **Ping** and **Ipconfig** return familiar-looking results.

In some cases, the commands that you are running really are the same old commands that you have probably run hundreds of times. For example, **Ping.exe** and **Ipconfig.exe** are *external commands*. This means that Windows PowerShell is running the same exact commands that you have always used.

In other cases, the commands are not exactly same. **Dir** and **Ls**, for example, are actually *aliases*, or nicknames, for the native Windows PowerShell cmdlet **Get-ChildItem**. Many built-in aliases are available to make Windows PowerShell commands available under familiar names, or to shorten long command names for easier typing. But these aliases do not change the way that the commands work—they just provide another name by which you can access the command. Try running **dir /s**, for example, and you will not get a directory that includes subfolder recursion. You will probably receive an error, instead. That

- Familiar-seeming commands
  - Dir
  - Cd
  - Mkdir
  - Type
- These are really *aliases* to PowerShell commands
- External commands like Ping.exe and Ipconfig.exe all work as usual
- PowerShell commands often have a different syntax, even if accessed by an alias that matches an older command name

is because the **Get-ChildItem** command does not offer **/s** as a command-line parameter; it uses a different syntax that you must learn.

## Learning Command Syntax

Windows PowerShell provides extensive in-product Help for commands that can be accessed through the **Get-Help** command or one of its shortcut functions, **Help** and **Man**. All three return basically the same result on-screen. The latter two default to displaying content in a page-at-a-time format in the console application, whereas **Get-Help** displays all content and lets you scroll backward to view it.

Running **help dir** will display the Help for **Get-ChildItem**, because that is the command actually run by the **Dir** alias. After briefly reviewing that help, you could learn that the command offers the **-recurse** command-line parameter to duplicate the functionality of the old **/s** switch.

- **Get-Help** (or **Help** or **Man**) provides quick access to syntax help for PowerShell commands
- Asking for help on an alias displays the help for the underlying command
- Use **-ShowWindow** to display help in a floating window
- Use **-Example** to quickly jump to usage examples
- Use **-Online** to display web-based version of help in your system's default web browser

Viewing the Help in the console window (or the console pane of the ISE) can sometimes be less-than-satisfactory. Try running **help dir -ShowWindow** to display the Help in a separate, floating window, which could even be moved to a second monitor. This makes it easier to read the Help while typing your command.

One of the most common reasons to seek out help is to find usage examples for a command, and Windows PowerShell typically includes many such examples. For example, run **help dir -example** to see examples of using the **Get-ChildItem** command. These can also be viewed near the end of the Help file when you are using the **-ShowWindow** option.

Finally, understand that Help files may occasionally contain errors or omissions. They are, after all, the product of human beings! Windows PowerShell offers two methods for obtaining the most recent Help content. First, the updatable Help system is capable of downloading updated Help to your computer, and you will explore that capability later in this module. Second, you can view online versions of the Help files. Because these are published online, Microsoft can publish updates with whatever frequency is necessary. For example, to view the online Help for **Get-ChildItem**, run **help Get-ChildItem -online**. Be aware that this requires an Internet connection and will display Help in your computer's default web browser.

## Demonstration: Viewing Help

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

### Demonstration Steps

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

## Finding Commands

Because Windows PowerShell has extensive built-in help, you can frequently learn how to use a command fairly quickly. The inclusion of usage examples further facilitates new command usage. The trick frequently is, then, to *find* the command in the first place. What command would you use to set an IP address on a network adapter? What command displays mailboxes in Exchange Server 2010? What command disables an Active Directory® user account?

The Help system, together with the **Get-Command** command, is actually designed to facilitate command discovery.

- Command naming convention: *Verb-SingularNoun*
- Given the naming convention and some experience, you can begin guessing command names
- Use **Help** and **Get-Command** with wildcards to validate your guesses and find commands
- After you find a command that looks like it will do what you need, read its complete help to learn how to use it
- No match using a **\*wildcard\*** search? Help will do a full-text search of selected Help file content
- You can also list the commands from a particular add-in module

 **Note:** Windows PowerShell uses the generic term *command* to refer to cmdlets, functions, workflows, applications, and other items. There are some differences in how these different items are created, but for now you can consider them to all work in the same way.

First, be aware that much of Windows PowerShell's functionality comes from add-in *modules*, each of which can add more commands to the shell's capabilities. When it is stored in the correct location (which you will explore later in this course), Windows PowerShell can detect these modules even when they are not loaded into memory, and can inventory their available commands. This *module discovery* feature enables Help and **Get-Command** to help you discover any command installed on your computer, even if the module that contains that command hasn't yet been loaded into memory. When you run a command whose module hasn't yet been loaded into memory, *module autoloading* will implicitly load the module. This makes the command available for execution.

The **Get-Help** command accepts *wildcard characters*, notably the asterisk (\*) wildcard character. When you ask for help and use wildcard characters with a partial command name, the shell will display a list of matching Help topics. This helps you find commands by making guesses.

So where do you start guessing?

Windows PowerShell's native commands—that is, those built into the shell and added in through modules—follow a strict naming convention that consists of a verb (or a verb-like word) and a singular noun. Listing processes, for example, is performed by running **Get-Process**. Listing services is performed by **Get-Service**. Creating a new service is performed by **New-Service**. The list of verbs is managed by Microsoft and can be viewed by running **Get-Verb**. As soon as you become familiar with this naming convention, you can start to make good guesses about command names. Do you need to display a list of mailboxes in Exchange Server? **Get-Mailbox** might be a good guess, and you could validate that by running **Help \*mailbox\***.

 **Note:** The Command Pane in the ISE application can also be used to find commands. It lists all installed commands alphabetically, and lets you type a partial command name to see matching commands.

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

**Get-Command** also accepts wildcard characters. This means that you can run **Get-Command \*event\***. This command also has two parameters that let you specify a specific verb or noun. For example, run **Get-Command –Noun event\*** or **Get-Command –Verb Get** to see a list of commands whose nouns start with **event** or a list of commands that use the **Get** verb.

 **Note:** Not all terms identified as verbs are actual English verbs. For example, *New* is not a verb, but it is the word Windows PowerShell uses to describe the operation of creating something new. Windows PowerShell uses the term *verb* somewhat loosely in this respect.

Sometimes, you may specify a wildcard search that cannot be fulfilled by a command name. For example, running **Help \*beep\*** will not find any commands that have *beep* in their name. So, the Help system will conduct a full-text search of available command *descriptions* and *synopses*. This would locate any Help files that contain *beep*. If only a single file is found with a match, the Help system will display it instead of showing a one-item list.

### **Listing the Commands from a Particular Add-in Module**

You may sometimes just want to see the commands from a specific module. For example, if you discovered the *NetAdapter* module, the name of which implies it works with network adapters, you may want to see which commands are in it.

You can do this by running (for example) **Get-Command –Module NetAdapter**. The **–Module** parameter restricts the command listing to just those commands in the designated module. You will frequently notice that the commands in a given module all share a common, short prefix on the noun part of their command name. For example, commands in the *NetAdapter* module all have nouns that begin with **NetAdapter**. These noun prefixes Help command names to remain unique across modules and help better identify the specific technology or component that the commands work with.

You can access a list of all installed modules by running **Get-Module –ListAvailable**. With the module names in front of you, you can easily list the commands in each module, and use **Help** to learn how to use each command.

## **Demonstration: Finding Commands**

In this demonstration, you will see several techniques that can be used to discover new commands. In the demonstration steps, italic text is sometimes used to help give you clues as to keywords that you might use with **Help**, **Get-Command**, and wildcard characters.

### **Demonstration Steps**

1. Find a command that could convert content to HTML.
2. Show a list of commands that can create new items. Notice that Windows PowerShell uses the word *new* as a verb, although in English the word is not actually a verb.
3. Find a command that could restart a computer.
4. Show a list of commands that deal with IPv4 addresses.
5. There is a command able to read Windows Event Logs (actually, there are two). Can it do so from a remote computer in addition to the local one?

## Interpreting the Help

As soon as you find the command that you must have for a particular task, you can use its Help file to learn how to use it. Although the examples are obviously a great way to do that, learning to interpret the Help file syntax can enable you quickly decipher a command's capabilities so that you can run it more easily.

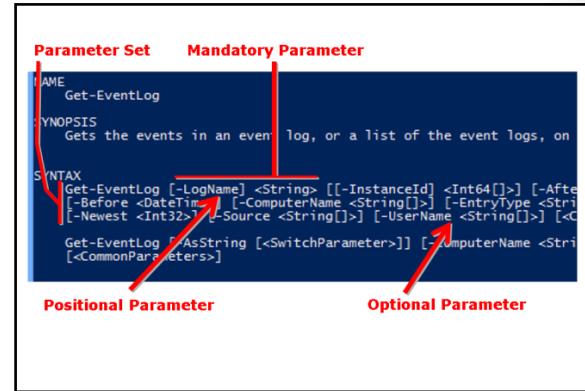
You will use the Help for **Get-EventLog** as an example.

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

In the first parameter set, the **-LogName** parameter is mandatory. This is indicated by the whole parameter not being enclosed in square brackets. The Help also shows that the parameter accepts **<string>** values meaning strings of letters, numbers, and other characters.

The actual **-LogName** parameter name is listed in square brackets, meaning it is a *positional parameter*. The command cannot be run without a log name. But you do not have to actually type the **-LogName** parameter name. You have to pass the log name string in the first position, because that is the position in the Help file where the **-LogName** parameter appears. Therefore, the following two commands are functionally equivalent:

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



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

Be cautious when omitting parameter names. One reason for caution is that you cannot do so with every parameter. The **-ComputerName** parameter, as one example, cannot have the parameter name omitted. Another reason for caution is that you can quickly lose track of what goes where. When you provide parameter names, the parameters can come in any order:

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

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

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



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

## Full Help

Although the brief syntax section of the Help file can be useful as a quick reminder, it does not provide a high level of detail about each parameter. Reading a command's full Help provides much deeper detail. For example, run **Get-Help Get-Service -Full** and see how much additional information you receive.

Some additional information includes the following:

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

Full Help also includes extended command descriptions and examples.

## Updating Help

Windows PowerShell 3.0 and later do not ship with Help files. Instead, Help files are provided as an online service. Microsoft-authored commands have their Help files hosted on a Microsoft-owned Web server; third-party commands can also use downloadable help, provided the author or vendor builds the module correctly and provides an online location for the Help files.

Updatable Help means that authors who write commands, including Microsoft, can make corrections and improvements to their Help files over time, and can deliver those to you without having to create an entire product update.

- No Help content is distributed with PowerShell
- **Update-Help** utilizes downloadable help content to update your local Help
- Checks no more than once every 24 hours by default
- **Save-Help** enables you to download Help and save it to an alternate location accessible to computers that are not connected to the Internet

Run **Update-Help** to scan your computer for all installed modules, retrieve online Help locations from each, and try to download Help files for each. You must run this command as a member of the local Administrators group, because Windows PowerShell core command Help is stored in the %systemdir% folder. Be aware that error messages will be displayed if Help cannot be downloaded for any locally installed module or modules. When Help cannot be downloaded, Windows PowerShell will still create a default Help display for the commands in the affected module.

Windows PowerShell defaults to downloading Help files in whatever language your system is configured to use. If Help is not available in that language, Windows PowerShell defaults to the **en-US** (US English)

language. You can override this behavior by using a parameter of **Update-Help** to specify the culture for which you want to retrieve help.

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

 **Note:** You should be aware that your lab virtual machines may not be connected to the Internet and thus may be unable to run **Update-Help** successfully. For your convenience, the virtual machines were populated with the latest Help content at the time that they were created.

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

Remember that both **Update-Help** and **Save-Help** will download Help only for modules that are installed on the computer where the command is run. They will not download Help for modules that are located on other computers.

## "About" Files

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

You can see a complete list of these topics by running **help about\***, and then view a single topic by running **help topicname**, such as **help**

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

When you use wildcard characters with the **Help** command, "About" Help files will be listed when their titles contain a match for your wildcard pattern. Typically, "About" Help files will be listed last, after any commands whose names also matched your wildcard pattern.

For much of the rest of this course, you will be referred to these "About" files for additional documentation. Frequently, you must read these files to discover the steps and techniques needed to complete lab exercises.

- Provide documentation for global shell techniques, concepts, and features
- Start with **about\_**
- View list by running **help about\***
- Keep these in mind: you will need to read many of these files to complete several upcoming lab exercises

## Demonstration: Using "About" Files

In this demonstration, you will see how to use the "About" Help file topics. Remember that these are compatible with Windows PowerShell's full-text search functionality, which is engaged when you ask for

MCT USE ONLY. STUDENT USE PROHIBITED

Help by using wildcard characters (such as by running **help \*processor\***) and the Help system does not find any command names that match your wildcard criteria.

### Demonstration Steps

1. View a list of "About" topics.
2. View the **about\_aliases** Help topic.
3. View the **about\_eventlogs** Help topic in a window.
4. View a Help topic that will explain how to make the console beep.

**Question:** Is there a way to specify multiple keywords when you search Help?

## Lesson 3

# Running Commands

Now that you have some familiarity with how to find commands and learn their syntax, you can start to run commands. Paying extra attention to syntax will help you avoid many of the challenges that Windows PowerShell newcomers frequently experience.

### Lesson Objectives

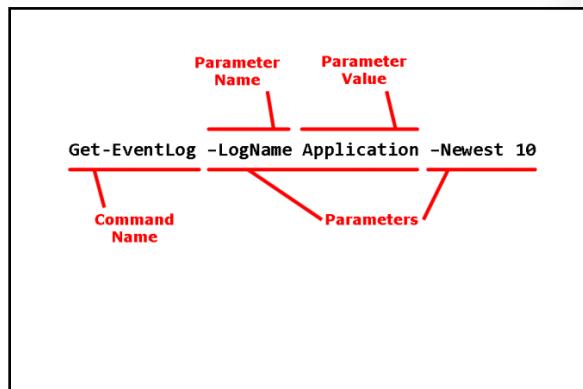
After completing this lesson, students will be able to:

- Execute commands by using the full command syntax
- Provide multiple values to a parameter
- Execute commands by using shortened command syntax
- Execute the **Show-Command** command
- Execute commands that change the system configuration

### Full Command Syntax

You have already seen several examples of Windows PowerShell commands being run, but now is a good time to focus on the full, formal syntax of these commands.

Each command starts with the command name, or an *alias*. Remember that command names (but not necessarily aliases) take the form *Verb-Noun*, such as **Get-Service**. Verbally, most Windows PowerShell users do not voice the dash and pronounce the command as "Get Service." Never forget that the dash is in there! Running **Get Service** will not return the results that you expect.



Following a command are zero or more parameters. In the full syntax, each parameter starts with a dash or hyphen, followed by the parameter name, a space, and then the parameter's value:

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

You should be aware that Windows PowerShell is not case-sensitive in most cases; typing a command name and its parameter names (and even usually the values) in all-lowercase will not make a difference.

You can use tab completion in both the console and the ISE applications to make typing easier, to avoid typos, and to double-check the syntax of the command. For example, try the following, pressing the Tab key every time that you see **[tab]**:

```
Get-EventL[tab] -Lo[tab] App[tab] -New[tab] 10 -Comp[tab] LON-DC1
```

Command names, parameter names, and even some parameter values can be completed in this manner.

Do not be confused about parameter names! The following is a common mistake:

```
Get-EventLog -Application
```

Here, the user probably meant to specify **-LogName Application** but has become confused. Merely providing **Application** without the dash would also have worked, because **-LogName** can accept a value by position. Here, the user has mixed up the two techniques, and specified a nonexistent parameter - **Application** without a value.

### Pay Attention to Spaces

Windows PowerShell uses the space character as a delimiter. It separates a command from its parameters, and separates parameters from their values. Where the shell is expecting a space, you can actually insert as many as you want to without breaking the command. However, you cannot add spaces anywhere where the shell is not expecting it. For example, all the following are incorrect:

```
Get EventLog -LogName Application
Get-EventLog - LogName Application
Get Event Log Application
```

These all add spaces where the shell is not expecting them, and the commands will not work.

## Specifying Multiple Parameter Values

Some parameters accept more than one value. In the Help file Syntax section, these are designated by a double-square-bracket notation in the parameter value type. For example:

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

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

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

Another way to specify multiple values is with a *parenthetical command*. This is a technique you will explore in more detail later, but generally it works as follows: suppose that you have a text file that lists one computer name per line:

```
LON-CL1
LON-DC1
```

If that file was named C:\computers.txt, you could use it in a parenthetical command as follows:

```
Get-EventLog -LogName Application -ComputerName (Get-Content C:\computers.txt)
```

 **Note:** **Get-Content** is the command behind the aliases **Type** and **Cat**. It reads the content from the text file and returns each line of the text file as a separate item.

A final technique would be to put the content into a variable, and pass the variable to the parameter. You will learn more about variables in upcoming modules. But an example of the correct syntax looks as follows:

```
$names = Get-Content C:\computers.txt  
Get-EventLog -LogName Application -ComputerName $names
```

## Shortened Command Syntax

Windows PowerShell's designers knew that they were giving us many long command and parameter names, and did not want us to have to spend more time than necessary typing them all out. Tab completion is one way to make your life easier, but Windows PowerShell also supports shortened command syntax through several techniques:

- Aliases can be used instead of command names. Aliases are generally shorter, although they are also frequently more cryptic and more difficult to read for someone not familiar with them.
- Parameter names can be omitted entirely for positional parameters. Instead, pass values in the appropriate command-line positions.
- All parameter names can be *truncated*, or shortened. You have to type only enough of each parameter name for the shell to be able to *disambiguate*. That is, you have to type enough characters so that the shell can determine the single parameter that you mean.

For example, the following two commands are functionally equivalent:

```
Get-Service -Name BITS -ComputerName LON-DC1  
gsv BITS -Comp LON-DC1
```

The second version is obviously easier to type, but it also somewhat more cryptic and more difficult to read.



**Note:** Instead of truncating or omitting parameter names, consider using tab completion instead. If you have already typed enough of the parameter name for the shell to disambiguate, you just have to press Tab to complete the name. The complete parameter name will always be easier to read for someone else who is trying to understand or maintain your commands.



**Best Practice:** In any form of permanent storage, whether a script file or an Internet blog, try to use the full command syntax. Use full command names, full parameter names, and names for all parameters. If you do this, it makes your commands easier for other people to read and understand.

MCT ISE ONLY STUDENT USE PROHIBITED

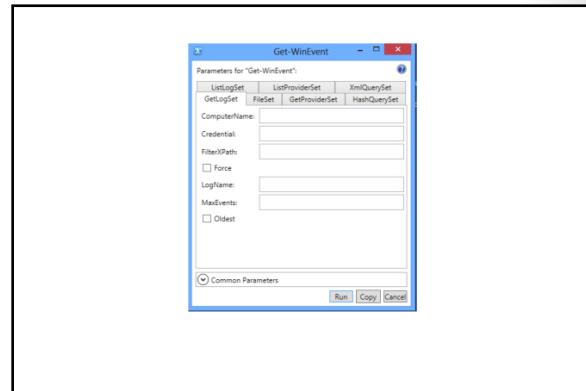
## Show-Command

**Show-Command** is a special Windows PowerShell command that accepts a single command name and then displays a graphical dialog box that has that command's parameters. Each parameter set is shown on a separate tab. This makes it visually clear that parameters cannot be mixed and matched between sets. To start the dialog box, specify the command name that you want to see:

```
Show-Command -Name Get-WinEvent
```

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

```
Show-Command Get-WinEvent
```



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

As soon as you fill in the desired parameter values, you can run the command immediately or copy it to the Clipboard. After copied, it can be pasted into the shell so that you can see the correct command-line syntax without running the command.

The Command Pane in the ISE application provides similar functionality. After you have found the command that you want in the Command Pane's list, you click that command to switch to a view very much like that of the **Show-Command** dialog box. The completed command can be run or copied into the ISE's active pane.

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

## Demonstration: Using Show-Command

In this demonstration, you will see how to use **Show-Command** to complete and run a Windows PowerShell command.

### Demonstration Steps

1. Display the list of parameters available for use with the **Get-Service** cmdlet using the command **Show-Command**.
2. Select a parameter set and fill in parameter values
3. Run the command, or copy it to the Clipboard and paste it into the shell for review.

## Commands that Modify the System

Many Windows PowerShell commands change the system state or configuration in some way. Actions might include restarting a computer, modifying a service, disabling a user account, and so on. Windows PowerShell standards specify that such commands support two parameters, **-WhatIf** and **-Confirm**, which enable you to test and control command execution. Commands that do support these parameters will list the parameters in their Help files.

When it is run with **-WhatIf**, a command will not actually execute. Instead, it will display messages that indicate what it would have done. When it is run with **-Confirm**, the command will stop and ask you whether it should continue for each item that it planned to modify. For example, if you are trying to stop 10 services, you would be prompted 10 times, once for each service.

Each command has an internal setting called its *confirm impact*. This is set by the command's developer to **Low**, **Medium**, or **High**. Windows PowerShell has two built-in *preference variables*, or settings, that interact with the confirm impact and the confirmation behavior. These preference variables are named **\$ConfirmPreference** and **\$WhatIfPreference**, and when you open a new shell session, they are each set to **High** by default. You may change them to **Low** or **Medium**. For example:

```
$ConfirmPreference = "Medium"
```

You should be aware that your changes affect only the current shell session; other open sessions are not affected, and your change is lost when you close the current session.

Windows PowerShell's confirmation behavior works as follows: when the command is run, Windows PowerShell checks its internal confirm impact level. If that level is equal to or higher than the appropriate preference variable, Windows PowerShell performs the confirmation action automatically.

For example, suppose that a command has a confirm impact of **Medium**, and you set **\$ConfirmPreference** to **Low**. When the command runs, the shell will see that its impact is higher than the preference and will act as if you had used **-Confirm**. This means that you will be prompted to continue or to halt the command.

If a command has an internal confirm impact of **High**, that command will always autoconfirm. You can override this by specifying **-Confirm:\$false**.

### Demonstration: Using **-WhatIf** and **-Confirm**

In this demonstration, you will see how to use the **-WhatIf** and **-Confirm** parameters, and the **\$ConfirmPreference** preference variable.

#### Demonstration Steps

- Run **Stop-Service** by using **-WhatIf** on the BITS service.
- Run **Stop-Process** by using **-confirm** on **Notepad**.
- Modify **\$ConfirmPreference** and see how it modifies command behavior.
- View the **-WhatIf** and **-Confirm** prompts in command Help for the **Clear-EventLog** cmdlet.

**Question:** Are **-WhatIf** and **-Confirm** supported by all commands that modify the system state or configuration in some way?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Finding and Running Basic Commands

### Scenario

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

### Objectives

After completing this lab, students will be able to:

- Discover new Windows PowerShell commands
- Execute basic Windows PowerShell commands
- Use Windows PowerShell "About" topics to learn new shell concepts and techniques

### Lab Setup

Estimated Time: 45 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Finding Commands

#### Scenario

In this exercise, you will use Windows PowerShell's **Help** (or **Get-Help**) and **Get-Command** commands to discover new commands capable of completing specific tasks within the shell.

In your tasks, *italicized* terms are intended to be keyword clues to help you complete the task.

There is only one task for this exercise:

- Find commands that will accomplish specified tasks

MCT USE ONLY. STUDENT USE PROHIBITED

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

On the 10961B-LON-CL1 virtual machine, log on as **Adatum\Administrator** and determine answers to the following questions:

1. What command would you run to resolve a DNS name?
2. What command would you run to make changes to a network adapter? After finding a command that would make changes to a network adapter, what parameter would you use to change its MAC Address (on adapters that support having their MAC address changed)?
3. What command would let you enable a previously disabled scheduled task?
4. What command would let you block access to a file share by a particular user?
5. What command would you run to clear your computer's local **BranchCache** cache?
6. What command would you run to display a list of Windows Firewall rules? What parameter of that command would display only enabled rules?
7. What command would you run to display a list of all locally bound IP addresses?
8. What command would you run to suspend an active print job in a print queue?
9. What native Windows PowerShell command would you run to format a new disk volume?

**Results:** After completing this exercise, you will have demonstrated your ability to use the command discoverability features of Windows PowerShell™ to find new commands that perform specific tasks.

## Exercise 2: Finding and Running Commands

### Scenario

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

There is only one task for this exercise:

- Run commands to accomplish specified tasks

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

1. On the 10961B-LON-CL1 virtual machine, while logged on as Adatum\Administrator, determine answers for the following questions:
  2. Display a list of enabled Windows Firewall rules.
  3. Display a list of all local IPv4 addresses.
  4. Set the startup type of the BITS service to **Automatic**.
1. Open the Computer Management console and go to Services and Applications.
2. Locate the Background Intelligence Transfer Service (BITS) and note its startup type setting prior to and after changing the startup type in Windows PowerShell.
3. Test the network connection to the computer named LON-DC1. Your command should return only a True or False value, without any other output.
4. Display the newest 10 entries from the local Security event log.

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

## Exercise 3: Using "About" Files

### Scenario

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

Words in italic are intended as clues. Remember that you have to use **Help** (or **Get-Help**) and wildcard characters. Because "About" files are not commands, **Get-Command** will not be useful in this exercise.

The main tasks for this exercise are as follows:

1. Locate and read "About" Help files
2. To prepare for the next module

#### ► Task 1: Locate and read "About" Help files

1. Ensure you are still on the 10961B-LON-CL1 virtual machine logged on as Adatum\Administrator from the previous exercise, and answer the following questions:
  2. What comparison operator does Windows PowerShell use for wildcard string comparisons?
  3. Are Windows PowerShell comparison operators typically case-sensitive?
  4. How would you use **\$Env** to display the **COMPUTERNAME** environment variable?
  5. What external command could be used to create a self-signed digital certificate usable for signing Windows PowerShell scripts?

#### ► Task 2: To prepare for the next module

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

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

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

**Question:** What is the main difference between **Get-Help** and **Get-Command**?

## Module Review and Takeaways



**Best Practice:** When you discover a new command, either by using **Help** or **Get-Command**, or by reading about the command someplace, always take a moment to read the command's Help file and learn a bit about its additional capabilities.



**Best Practice:** Even familiar commands can gain new functionality in new versions of Windows PowerShell. Take several minutes to read the Help files even of commands that you already know well from earlier versions, to see what new features may exist.

### Common Issues and Troubleshooting Tips

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

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 2

## Working with the Pipeline

### Contents:

Module Overview	02-1
<b>Lesson 1:</b> Understanding the Pipeline	02-2
<b>Lesson 2:</b> Selecting, Sorting, and Measuring Objects	02-6
<b>Lab A:</b> Using the Pipeline	02-13
<b>Lesson 3:</b> Converting, Exporting, and Importing Objects	02-16
<b>Lab B:</b> Converting, Exporting, and Importing Objects	02-21
<b>Lesson 4:</b> Filtering Objects Out of the Pipeline	02-24
<b>Lab C:</b> Filtering Objects	02-30
<b>Lesson 5:</b> Enumerating Objects in the Pipeline	02-33
<b>Lab D:</b> Enumerating Objects	02-36
Module Review and Takeaways	02-38

## Module Overview

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



**Additional Reading:** You can read more about Understanding the Windows PowerShell Pipeline at <http://go.microsoft.com/fwlink/?LinkId=306145>.

### Objectives

After completing this module, students will be able to:

- Describe the purpose of the Windows PowerShell pipeline
- Manipulate objects in the pipeline
- Convert, export, and import objects
- Filter objects out of the pipeline
- Enumerate objects in the pipeline

## Lesson 1

# Understanding the Pipeline

In this lesson, you will learn about the Windows PowerShell pipeline. You will also learn the basic techniques for running multiple commands in the Windows PowerShell pipeline.

### Lesson Objectives

After completing this lesson, students will be able to:

- Describe the pipeline
- Execute basic commands in the pipeline
- Describe pipeline output and objects by using appropriate terminology
- Discover and display object members
- Explain the possible problems of running commands that produce more than one kind of object

### What Is the Pipeline?

Windows PowerShell runs commands in a *pipeline*. A pipeline can contain one command or multiple commands. For multiple commands, each command, and its parameters, are separated by a vertical pipe (|) character. The commands in the pipeline run in sequence from left to right. As each command runs, its output can be passed to the next command, which can use that output as part of its own behavior. Specific rules dictate how output is passed from one command to the next. You will learn about those rules in Module 3, "Understanding How the Pipeline Works."

- Windows PowerShell runs commands in a *pipeline*
- In the console, each complete command line is a pipeline
- Pipelines can contain one or more commands, with multiple commands separated by a vertical pipe character (|)
- Commands execute from left to right, with the output of each command being *piped* (passed) to the command after it
- The output of the last command in the pipeline is what appears on your screen

As you interact with Windows PowerShell in the console host application, you can think of each command line as a single pipeline. You type one or more commands and press Enter to run the pipeline. The output of the last command in the pipeline appears on your screen. That output is followed by another shell prompt, at which you can enter commands into a new pipeline.



**Note:** You can type a single logical command line over multiple physical lines in the console. For example, type **Get-Service** ' and press Enter. The shell will enter an extended prompt mode, indicated by the presence of the symbols >>. This enables you to complete the command line. For now, press Ctrl+C to exit the command and return to the shell prompt.

## Running Commands in the Pipeline

You have already been running commands in the pipeline, although you have probably been running just one command at a time, such as **Get-Service**.

Running multiple command pipelines is similar. For example, try running **Get-Service | Out-File ServiceList.txt**. That pipeline contains two commands. The output of **Get-Service** is piped to **Out-File**. One of the parameters of **Out-File** specifies the file name that the output should be sent to. When you run that command, nothing appears on the screen because **Out-File** itself does not generate any output in the pipeline. It does create a file on disk, but because it does not explicitly write anything to the shell pipeline, nothing is displayed on the screen.

 **Note:** **Out-File** will create a text file that has contents that match what would have otherwise appeared on the screen. As an alternative, Windows PowerShell lets you use the greater than symbol (>) instead. For example, **Get-Service > ServiceList.txt** will create a service list in a text file that is named ServiceList.txt.

## Pipeline Output

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

You can imagine command output looking like a table, or a Microsoft® Office Excel spreadsheet. In Windows PowerShell terminology, the table or spreadsheet is a collection of objects, or just *collection* for short. Each row is a single *object*, and each column is a *property*. When you run **Get-Service**, the command produces a collection of service objects. Each has properties like **Name**, **DisplayName**, **Status**, and so on.

Windows PowerShell's use of objects is very different from other command-line shells whose commands primarily generate text. In a text-based shell, suppose that you wanted to obtain a list of all services that were started. You might run a command to produce a text list of services. You would then pipe that text to a second command by using parameters to define a particular column position that contains status information. If the output of the first command ever changed, and the status information moved, you would have to rewrite the second command to have the new position information. Text-based shells frequently require great skill with text parsing. This makes scripting languages such as Perl popular, because it offers strong text parsing and text manipulation features.

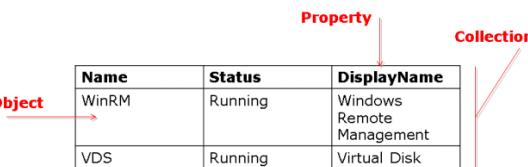
In Windows PowerShell, you would just tell the shell to produce a collection of service objects, and to then display only the **Name** property. The structure of the objects in memory enables the shell to find the information for you, instead of you having to worry about the exact form of the command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

- **Get-Service** is a single-command pipeline.

- Multicommand pipelines look similar:  
**Get-Service | Out-File ServiceList.txt**

- The preceding pipeline produces no output on the screen. Why?

- Windows PowerShell commands usually produce *objects* as their output
- Think of these as a table of data in memory



Name	Status	DisplayName
WinRM	Running	Windows Remote Management
VDS	Running	Virtual Disk

## Discovering Object Members

The properties of an object are just one of its members. The word *members* describes various components of an object, including the following:

- Properties that describe attributes of the object. A service name, a process ID number, and an event log message are all examples of properties.
- Methods that tell an object to perform an action. A process object could quit itself, and an event log could clear itself.
- Events that are triggered when something happens to an object. A file might trigger an event when it is opened, or a process might trigger an event when it has output to produce.

Object members include:

- Properties
- Methods
- Events

Run a command that produces an object, and pipe that object to **Get-Member** to see a list of members

**Get-Member** is a discovery tool, similar to **Help**, that can help you learn to use the shell

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

You can use the **Get-Member** command to list all the members of an object. It will list all properties, even those that are not displayed on the screen by default. This command will also list methods and events and show you the type name of the object. For example, the objects produced by **Get-Service** have the type name **System.ServiceProcess.ServiceController**. You can use the type name in Internet search queries to locate object documentation and examples (although those examples will frequently be in a programming language such as Microsoft Visual Basic® or Microsoft Visual C#®).

 **Note:** **Get-Member** has an alias, **Gm**.

To use **Get-Member**, just pipe any command output to it:

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

 **Note:** The first command will actually run, produce its output, and then pass that output to **Get-Member**. Use caution when you run commands that may modify the system configuration, because those commands will actually be running. You cannot use the **-WhatIf** parameter on any command that you want to pipe to **Get-Member**, because the parameter prevents the command from producing any output. That means **Get-Member** is given no input. Therefore, it will display no output.

## Demonstration: Pipeline Basics

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

### Demonstration Steps

- Display a list of services.

2. Display a list of processes.
3. Write a list of services to a file that is named ServiceList.txt.
4. Find a command that can display output in a grid view.
5. Display the most recent five entries from the Security event log in a grid view.
6. Determine the type name of the objects produced by **Get-Process**.
7. For the objects produced by **Get-NetAdapter**, determine which property lists the adapter's maximum speed.

## When the Pipeline Contains Mixed Output

Although most Windows PowerShell commands produce just one kind of object, some commands are capable of producing multiple kinds of objects. For example:

```
Get-ChildItem -Path C:\Windows | Get-Member
```

This command produces two kinds of objects: **FileInfo** and **DirectoryInfo**. It does so because a single directory listing can include both files and folders (directories).

- Most commands produce just one kind of object
- Some commands produce more than one kind
- When working with more than one kind of object in the pipeline, use extra caution
  - Not every kind of object has the same members
  - Output might not be what you originally expected
- **Get-Member** can handle multiple kinds of objects and will display each kind in a separate list

Commands that produce multiple kinds of objects can be more difficult to work with and require extra thought from you. For example, a **FileInfo** object has a **Length** property that contains the file size in bytes. A  **DirectoryInfo** object does not have a corresponding property. If you were writing a command line that first ran **Get-ChildItem** (or one of its aliases, such as **Dir** or **Ls**), and then tried to filter the output based on the **Length** property, you might not obtain the results that you expected because  **DirectoryInfo** objects lack the property. Always be careful when the pipeline contains multiple kinds of objects.

 **Note:** If you pipe multiple kinds of objects to **Get-Member**, the command will display the members for each kind of object separately. For example, piping **Get-EventLog** to **Get-Member** can produce a large amount of output because each kind of event log entry is seen by **Get-Member** as a different kind of object.

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

MCT USE ONLY  
STUDENT USE PROHIBITED

## Lesson 2

# Selecting, Sorting, and Measuring Objects

In this lesson, you will learn to manipulate objects in the pipeline by using commands that sort, select, and measure.

## Lesson Objectives

After completing this lesson, students will be able to:

- Sort objects on a specified property
- Measure objects' numeric properties
- Display a subset of objects in a collection
- Display a customized list of objects' properties
- Create custom properties for display

## Sorting Objects on a Property

Some Windows PowerShell commands produce their output in a specific order. For example, both **Get-Process** and **Get-Service** produce output that is sorted alphabetically by name. **Get-EventLog** produces output sorted by time. Sometimes, you may want command output sorted differently. The **Sort-Object** command (which has the alias **Sort**) can do that for you.

The command accepts one or more property names to sort on. 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 on the first property first, and then on the second property, and so on. It is not possible in a single command to sort on one property in ascending order and another in descending order.

Some examples:

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

- Commands determine their own default sort order
- **Sort-Object** can re-sort objects in the pipeline
- Example of use:
  - **Get-Service | Sort-Object Name -Descending**

By default, string properties are sorted without regard to case. In other words, lowercase and uppercase letters are treated the same. Parameters of **Sort-Object** enable you to specify a case-sensitive sort, a specific culture's sorting rules, and other options.

## Demonstration: Sorting Objects

In this demonstration, you will see some ways to use **Sort-Object**.

### Demonstration Steps

1. Display a list of processes sorted by process ID.
2. Display a list of services sorted by status.
3. Display a list of the most recent 10 Security event log entries, sorted with the oldest entry first.
4. Display a list of services sorted first by status, then by name.

## 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.

The command has the alias **Measure**.

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

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

 **Note:** Because Windows PowerShell enables you to truncate parameter names, you will frequently see those written as **-Sum**, **-Min**, and **-Max**, corresponding to common English abbreviations for those words. However, **-Average** cannot be shorted to **-Avg**, although beginning users frequently try. The parameter could be shorted to **-Ave**, because that is a legal truncation of the name.

### Demonstration: Measuring Objects

In this demonstration, you will see how to use **Measure-Object**.

### Demonstration Steps

1. Display the number of services on your computer.
2. Display the number of running processes.
3. Display the total amount and average amount of virtual memory being used by processes.

## Selecting a Subset of Objects

Sometimes you may not need to display all the objects produced by a command. For example, you have already seen how the **Get-EventLog** command can produce 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 any command's output.

The command has the alias **Select**.

**Select-Object** can perform two separate tasks in one command. Selecting a subset of objects is just one of them. If you think of a collection of objects as a table or spreadsheet, selecting a subset means selecting just certain rows. With **Select-Object**, you cannot specify criteria to decide which rows will be selected. Instead, you can select only a specific number of rows from the beginning or end of the collection. You can also specify that a certain number of rows be skipped before the selection begins.

To select the first 10 processes based on lowest virtual memory use:

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

To select the last 10 running services sorted by name:

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

To select the 5 processes using the least amount of CPU, but skipping the one process using the least CPU:

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

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

## Selecting Properties of Objects

In addition to selecting the first or last number of rows from a collection, you can use **Select-Object** to select specified properties to display. If you think of a collection of objects as a table or spreadsheet, you are choosing the columns to display. After you choose the properties you want, **Select-Object** will remove all other properties. For example, if you want to sort on a particular property but not display it, you must use **Sort-Object** first, and then **Select-Object** to specify the properties that you want to display.

Use caution when specifying property names.

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

- This is the second use of **Select-Object**
- Use the **-Property** parameter to specify a comma-separated list of properties (wildcards are accepted) to include
- Can be combined with **-First**, **-Last**, and **-Skip** to select a subset of rows



**Best Practice:** Always review property names in the output of **Get-Member** before you use those property names in another command. Doing this guarantees that you are using the actual property name and not a made-up name created for display purposes.

To specify the properties to display, use the **-Property** parameter of **Select-Object**:

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

You can combine that parameter with **-First** or **-Last**:

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



**Note:** The output of that command might look odd. By default, Windows PowerShell tries to fill the width of your console window by spreading out table columns. When you display only some columns, the resulting output contains a large amount of space in between the columns. In Module 5, "Formatting Output," you will learn how to gain more precise control over the output display.

You can also specify a wildcard pattern for a property name, although if that results in lots of properties being selected, some of those properties may be truncated because of screen space:

```
Get-Process | Select-Object -Property *
```

## Demonstration: Selecting Objects

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

### Demonstration Steps

1. Display the 10 largest processes by virtual memory use.
2. Display the current day of week. For example, "Monday" or "Tuesday."
3. Display the 10 most recent Security event log entries. Include only the event ID, time written, and event message.

## Creating Calculated Properties

**Select-Object** can also create custom, or *calculated*, properties. Each of these 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. Each calculated property is typed in a hash table.

### The Hash Table

A *hash table* is known in some programming languages as an associative array, a dictionary, or a hash table. One hash table can contain multiple

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

items, and each item consists of a key and a value. Windows PowerShell uses hash tables many times and for many purposes. In some cases, you can specify your own keys. When a hash table is used to create calculated properties by using **Select-Object**, you must use the keys that Windows PowerShell expects. Those are as follows:

- **label, l, name, or n** to specify the label of the calculated property. Because the lowercase L resembles the number 1 in some fonts, try to use either **name, n, or label**.
- **expression or e** to specify the expression of the calculated property.



**Note:** This book will use **n** and **e** for most examples because those two use less space on the page.

For example, suppose that you want to display a list of processes that includes each process' name, ID, virtual memory use, and paged memory use. You want to use the column labels **VirtualMemory** and **PagedMemory** for the last two properties, and you want those values displayed in bytes. You could run the following command:

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



**Note:** If you type that command exactly as is, pressing Enter after the vertical pipe character, it will still work. You will enter extended prompt mode. After you type the rest of the command, press Enter on a blank line to execute the command.

That command includes two hash tables, each one creating a calculated property. **\$PSItem** is a special variable created by Windows PowerShell. It represents whatever object was piped into the **Select-Object** command. In this example, that is a **Process** object. The period after **\$PSItem** lets you access a single member of the object. In this example, one calculated property is using the **VM** property, and the other is using the **PM** property. The hash table can be easier to interpret if you write it out a bit differently:

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

As you can see, a semicolon separates the two key-value pairs. The keys **n** and **e** were used. These keys are expected by Windows PowerShell. The label (or name) is just a string. Therefore, it is enclosed in quotation marks. Windows PowerShell accepts either single or double quotation marks for this purpose. The expression is a small piece of executable code called a *script block*, and is contained within curly braces.



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

A comma separates each property from the others in the list. Name, ID, and both calculated properties are all separated by commas.

## Formatting Tips

You might want to modify the previous command to display memory values in megabytes (MB). Windows PowerShell understands the abbreviations **KB**, **MB**, **GB**, **TB**, and **PB** as representing the base-2 values

kilobyte, megabyte, gigabyte, terabyte, and petabyte, respectively. So 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 revised formatting that makes the command easier to read, this new 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. Unfortunately, the resulting values have several decimal places, which is unattractive.

Consider this revision:

```
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 new example uses Windows PowerShell's **-f** formatting operator. To the left of the operator is a string that tells Windows PowerShell what data to display. **{0:N2}** means to display the first data item as a number with two decimal places. To the right of the operator is the original mathematical expression. It is in parentheses to make sure that it executes as a single unit. You can type this command exactly as shown, press Enter on a blank line, and view the results.

The syntax in that example can be confusing because there are a lot of punctuation symbols. Start with the basic expression:

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

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

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

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

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

## Demonstration: Creating Calculated Properties

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

### Demonstration Steps

1. Display a list of the most recent commands run in the shell.
2. View the members of the objects output by that command.
3. Locate the properties that contain the time that the command started running and the time that it finished running.

4. Display the same list, adding a calculated property that shows the time that it took each command to run.
5. Display the same list, sorting the results by the time that it took each command to run. Show longest-running commands at the top of the list, and shortest-running commands at the bottom.

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

# Lab A: Using the Pipeline

## Scenario

You must produce several basic management reports that include specified information about the computers in your environment.

## Objectives

After completing this lab, students will be able to:

- Modify and sort pipeline objects

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name : ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Selecting and Sorting Data

### Scenario

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

The main tasks for this exercise are as follows:

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

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

1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
2. Using a keyword like **date**, find a command that can display the current date.
3. Display the members of the object produced by the command that you found in the previous step.
4. Display only the day of the year.

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

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

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

1. Using a keyword like **DHCP** or **scope**, find a command that can display a list of IPv4 DHCP scopes.
2. View the Help for the command.
3. Display a list of available IPv4 DHCP scopes on LON-DC1.
4. Display a list of available IPv4 DHCP scopes on LON-DC1. Include only the scope ID, subnet mask, and scope name.

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

1. Using a keyword such as **rule**, find a command that can display firewall rules.
2. Display a list of firewall rules.
3. View the Help for the command that displays firewall rules.
4. Display a list of firewall rules that are enabled.
5. Display a list of enabled firewall rules. Display only the rules' display names, the profile they belong to, their direction, and whether they allow or deny access. Sort the list in alphabetical order by display name.

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

1. Using a keyword such as **neighbor**, find a command that can display network neighbors.
2. View the Help for the command.
3. Display a list of network neighbors.
4. Display a list of network neighbors, sorted by state.
5. Display a list of network neighbors, sorted by state and displaying only the IP address and state.

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

1. Ping LON-DC1.
2. Ping LON-CL1.
3. Using a keyword such as **cache**, find a command that can display items from the DNS client cache.
4. Display the DNS client cache.
5. Display the DNS client cache. Sort the list by record name, and display only the record name, record type, and time to live.

MCT USE ONLY. STUDENT USE PROHIBITED

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

**Question:** Suppose that you wanted to produce output that included all of an object's properties except one. What would be the most efficient way to do that?

## Lesson 3

# Converting, Exporting, and Importing Objects

In this lesson, you will learn about Windows PowerShell's built-in features for converting data to different formats, and for exporting and importing data from and to external storage.

## Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of object conversion
- Redirect command output to a file
- Convert objects to a different form
- Export objects to external storage
- Import objects from external storage

## Converting Objects to Another Form

Windows PowerShell includes the ability to convert pipeline objects to other forms of data representation. For example, a collection of objects can be converted to an HTML page or HTML fragment, to Comma-Separated Values (CSV) format, or to XML.

PowerShell uses two distinct verbs for conversion: **ConvertTo** and **Export**. A command that uses the verb **ConvertTo**, such as **ConvertTo-HTML**, accepts objects as input from the pipeline, and produces converted data as output to the pipeline. In other words, the data remains in

Windows PowerShell. The data could be piped to another command that wrote the data to a file or manipulated it in some other way. For example:

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

A command that uses the verb **Export**, such as **Export-CSV**, performs two operations at the same time. It first converts the data, and then it writes the data to external storage that might be a file on disk. For example:

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

Export commands basically combine the functionality of **ConvertTo** with a command like **Out-File**. Export commands do not usually put any output into the pipeline. Therefore, nothing appears on the screen after an export command finishes running.

A key part of both operations is that the form of the data is changed to something like CSV, XML, HTML, or another form. The data is no longer contained in the structure referred to as objects, and is instead represented in another form entirely. When you convert it to another form, data is generally more difficult to manipulate. Converted data cannot be easily sorted, selected, measured, and so on.

To see a list of commands, run **Get-Command -Verb ConvertTo,Export**. The commands that you will use most frequently include the following:

- **ConvertTo-CSV**
- **ConvertTo-HTML**
- **ConvertTo-XML**
- **Export-CSV**
- **Export-CliXML**

 **Additional Reading:** Windows PowerShell includes only basic HTML conversion capabilities. If you are interested in producing more complex management reports by using HTML, read "Creating HTML Reports in Windows PowerShell," a free e-book available at [www.PowerShell.org/books](http://www.PowerShell.org/books).

## Piping Output to a File

Windows PowerShell's **Out-File** command can accept input from the pipeline and write that to a file. It can render objects into text, and uses the same technique that the shell uses to render objects into text for on-screen display. In other words, whatever you pipe into **Out-File** will be the same as what would have otherwise appeared on the screen. This behavior differs from converting or exporting the objects, because you are not changing the form of the objects. Instead, the shell is capturing what would have appeared on the screen.

- **Out-File** writes whatever is in the pipeline to a text file
- The text file will be formatted exactly as the same data would have appeared on the screen—there is no conversion to another form
- Unless the data has been converted to another form, the resulting text file will usually be suitable only for viewing by a person
- As you start to build more complex commands, you will need to keep track of what the pipeline contains at each step

Various parameters of **Out-File** enable you to specify a file name, to append content to an existing file, to specify character encoding, and so on.

**Out-File** is perhaps the easiest way to move data out of the shell and into external storage. However, because the text files that **Out-File** creates are usually intended for viewing by a person, reading the data back into the shell in any way that enables the data to be manipulated, sorted, selected, measured, and so on is frequently difficult or impractical.

**Out-File** does not produce any output of its own. This means that the command does not put objects into the pipeline. After you run the command, you should expect to see no output on the screen.

### Keep Track of What the Pipeline Contains

As you begin to write more complex commands in Windows PowerShell, you will have to become used to keeping track of the pipeline's contents. As each command in the pipeline runs, the command may produce different output from what was put into it. Therefore, the next command will be working with something different. For example:

```
Get-Service |  
Sort-Object -Property Status |  
Select-Object -Property Name,Status |  
ConvertTo-CSV |  
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 the 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 is 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 the 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 **Name** and **Status** properties, so you could no longer sort them on their **DisplayName**, because that property no longer exists.
- After **ConvertTo-Csv** runs, the pipeline contains **System.String** objects that contain the CSV-formatted data. Windows PowerShell could no longer sort or select these objects, because they are no longer in a dedicated kind of object.
- After **Out-File** runs, the pipeline contains nothing. Therefore, nothing will appear on the screen after running this complete command.



**Note:** When you have a complex, multiple command pipeline such as this one, you may have to debug it if it does not run correctly the first time. The best way to debug is to start with just one command, and see what it produces. Then add the second command, and see what happens. Continue to add one command at a time, verifying that each one produces the output you expected before you add the next command.

## Demonstration: Converting and Exporting Objects

In this demonstration, you will see different ways to convert and export data.

### Demonstration Steps

1. Convert a list of processes to HTML.
2. Create a file that is named Procs.html that contains an HTML-formatted list of processes.
3. Convert a list of services to CSV.
4. Create a file that is 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.

## Importing Data

Importing is the process of reading formatted data from external storage, such as a file on disk, and converting that data back into objects. Those objects are usually put into the pipeline, where they can be passed to other commands for additional manipulation.

The effectiveness of the import process depends on the format that the data is in. CSV files, for example, are flat data structures and do not provide a high degree of fidelity for preserving complex data. By contrast, XML provides a higher-fidelity way of preserving data, even in complex hierarchical relationships.

You have already learned that the **Export** verb in Windows PowerShell implies a two-step process: data is first converted to another form, and then it is written to external storage. The **Import** verb also implies a two-step process: data is first read from external storage, and then it is converted from that format into objects. For example:

```
Get-Process | Export-Csv Processes.csv
```

This command saves a list of running processes in CSV format, in a file that is named `Processes.csv`. Here is the next command:

```
Import-Csv Processes.csv | Sort-Object -Property VM -Descending | Select-Object -First 10
```

This command reads that CSV file, reconstructs the original objects with some degree of fidelity, sorts those objects in descending order on their VM property, and then selects the first 10 objects.

Importing implies that the command understands the form that the data is in, and that the command can construct usable objects from that data. Importing differs from reading a file's contents and not paying attention to the form. For example:

```
Get-Content Processes.csv | Sort-Object -Property VM -Descending | Select-Object -First 10
```

This command will not produce the expected results. Instead of importing the data, the **Get-Content** command was used to read it. Using **Get-Content** resembles looking at a CSV file in Notepad, where you see the raw data. Importing is more like importing a CSV file into Microsoft Office Excel, where the file is interpreted and correctly broken out into various rows and columns in a spreadsheet.

Because **Get-Content** does not try to interpret the file, Windows PowerShell includes several commands that use the **ConvertFrom** verb. As the name implies, these commands are the opposite of those that use the **ConvertTo** verb. For example:

```
Get-Content Processes.csv | ConvertFrom-Csv | Sort-Object -Property VM -Descending | Select-Object -First 10
```

This command would have the result you want, because **Get-Content** is being used to read the file's raw contents, but then **ConvertFrom-CSV** is interpreting that data and constructing usable objects to put into the pipeline. Used together in this manner, **Get-Content** and **ConvertFrom-CSV** have the same result as **Import-CSV**.

- **Import** commands are the opposite of **Export** commands
- Two-step process to both read the raw data and convert it into usable objects
- **Import-Csv**, **Import-CliXML**, and so on
- The reconstructed objects may lack some of the members of the original objects, depending on the capabilities of the data format used



**Note:** In many cases, two commands might achieve the same result as one, but they may take longer to run. For example, **Import-CSV** alone runs faster than **Get-Content** and **ConvertFrom-CSV** combined.

## Demonstration: Importing Objects

In this demonstration, you will see different ways to read and import data.

### Demonstration Steps

1. Read the contents of Serv.csv without interpreting the data.
2. Import the contents of Serv.csv so that the data is interpreted into objects.
3. Using the services in Serv.csv, display a list of services that have running services listed first, stopped services listed second, and services sorted in reverse alphabetical order, by name, within those lists. Display only service names and status.

**Question:** What other data forms might you want to convert data to or from?

# Lab B: Converting, Exporting, and Importing Objects

## Scenario

You must convert management information into different formats for use by other people and processes in your environment.

## Objectives

After completing this lab, students will be able to:

- Convert objects to different forms
- Import objects from and export objects to external storage

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Converting Objects

### Scenario

In this exercise, you will convert objects to different forms of data. In some cases, you will write the converted data out to a text file.

The main tasks for this exercise are as follows:

1. Produce an HTML report listing the processes running on a computer
  - ▶ **Task 1: Produce an HTML report listing the processes running on a computer**
    1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
    2. Display a list of running processes.
    3. Display a list of running processes, sorted in reverse alphabetical order by process name that shows only the process name, ID, virtual memory, and physical memory consumption.

4. View the Help for **ConvertTo-HTML**.
5. Display the same list again, and convert the list to an HTML page.
6. Display the same list again, and convert the list to an HTML page. Store the HTML in ProcReport.html.
7. Use Windows Internet Explorer to view ProcReport.html.
8. Display the same list again, and convert it to HTML. Store the HTML in ProcReport.html, overwriting the existing file. Have the phrase *Processes* appear before the list of processes. Have the current date appear after the list of processes.
9. Use Internet Explorer to view ProcReport.html.

**Results:** After completing this exercise, you will have converted objects to different forms of data.

## Exercise 2: Importing and Exporting Objects

### Scenario

In this exercise, you will import objects from and export objects to files.

The main tasks for this exercise are as follows:

1. Create a comma-separated values (CSV) file listing the most recent 10 entries from the System event log.
2. Create an XML file listing services
3. Produce a pipe-delimited list of the most recent 20 Security event log entries
4. Import data from a pipe-delimited file

► **Task 1: Create a comma-separated values (CSV) file listing the most recent 10 entries from the System event log.**

1. Display the most recent 10 entries from the System event log.
2. Convert, but do not export, the event log list to CSV form.
3. Export the event log list to CSV form in a file that is named SysEvents.csv.
4. View SysEvents.csv in Notepad.
5. Create SysEvents.csv again, but remove the first line of the file that begins with #.
6. View SysEvents.csv in Notepad.

► **Task 2: Create an XML file listing services**

1. Display a list of services, with stopped services listed last.
2. Export the list to an XML file that uses the command-line interface (CLI) XML (CliXML) format. Name the file **Services.xml**.
3. View Services.xml in Notepad.
4. Create the list again, but include only services' names, display names, and status. Export the list to **CliXML** in the file Services.xml.
5. View Services.xml in Notepad.

► **Task 3: Produce a pipe-delimited list of the most recent 20 Security event log entries**

1. View the Help for **ConvertTo-CSV**.

2. Display a list of the most recent 20 Security event log entries.
3. Display the list again. Include only the event ID, time written, and message.
4. Export the list to a file that is delimited by using the vertical pipe (|) character. Name the file **Security.pdd**.
5. View **Security.pdd** in Notepad.

► **Task 4: Import data from a pipe-delimited file**

- Display the first 10 Security event log entries from the **Security.pdd** file that you created in the previous task.

**Results:** After completing this lab, you will have imported data from and exported data to external storage.

**Question:** Could you use **ConvertTo-CSV** or **Export-CSV** to create a file that was delimited by using a character other than a comma? For example, could you create a tab-delimited file?

**Question:** The HTML produced by **ConvertTo-HTML** looks very plain. The HTML standard offers a way to specify visual styles for an HTML document. This is known as a *cascading style sheet* (CSS). Does the command offer a way to attach a CSS?

## Lesson 4

# Filtering Objects Out of the Pipeline

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

### Lesson Objectives

After completing this lesson, students will be able to:

- List the major Windows PowerShell comparison operators
- Filter objects by using basic syntax
- Filter objects by using advanced syntax
- Optimize filtering performance in the pipeline

### Comparison Operators

To start filtering, you will need a way to tell Windows PowerShell which objects that you want to keep and which objects want to remove from the pipeline. You do this by specifying criteria for objects that you want to keep, and most of the time, you will do so by using one of the shell's comparison operators. You will ask the shell to compare some property of an object to some value that you specify, and if the comparison is true, the object is kept. If the comparison is false, the object is removed.

The basic comparison operators include the following:

- **-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

Be aware that for string comparisons, these operators are all *case-insensitive*, meaning they are not sensitive to case. A case-sensitive version of each is available if it is necessary, such as **-ceq** and **-cne**.

The shell 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.

Comparison	Case-InSensitive	Case-sensitive
Equality	<b>-eq</b>	<b>-ceq</b>
Inequality	<b>-ne</b>	<b>-cne</b>
Greater than	<b>-gt</b>	<b>-cgt</b>
Less than	<b>-lt</b>	<b>-clt</b>
Greater than or equal to	<b>-ge</b>	<b>-cge</b>
Less than or equal to	<b>-le</b>	<b>-cle</b>
Wildcard equality	<b>-like</b>	<b>-clike</b>

There are additional, more advanced operators that will not be covered at this point. These include the following:

- **-in** and **-contains** test to see whether an object exists in a collection.
- **-as** tests to see whether an object is of a particular type.
- **-match** and **-cmatch** compare a string to a regular expression.

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

Comparisons can be made directly at the command line prompt. For example:

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

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

## Basic Filtering Syntax

The **Where-Object** command (and its alias, **Where**) has two syntax forms. These two forms are created by having carefully designed parameter sets. This enables an easy-to-read syntax. For example, to display a list of only running services:

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

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

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

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

This command produces no output, because no service object had a **Stat** property that contained the value **Running**. Actually, none of the service objects have a **Stat** property at all. This is why the comparison returns False for every object.

- The **Where-Object** command provides filtering
- Basic syntax:

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

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

 **Note:** Because of the complex parameter sets needed to make the basic syntax functional, the Help file for **Where-Object** is very long and difficult to read. Consider skipping the initial Syntax section and going directly to the Description or Examples if you need help with this command.

## Limitations of the Basic Syntax

The basic syntax can be used with only a single comparison. You could not, for example, display a list of services that were stopped and had a start mode of automatic, because that would require two comparisons.

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

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

The intent is to display all 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 will have to move to the advanced filtering syntax.

## Advanced Filtering Syntax

The advanced syntax of **Where-Object** uses a filter script. Within that script, you can use the built-in **\$PSItem** variable (or **\$\_**, which is also valid in earlier versions of Windows PowerShell) to reference whatever object was piped into the command. Your filter script will execute one time for each object that is piped into the command. When your filter script returns True, that object is passed down the pipeline as output. When your filter script returns False, that object is removed from the pipeline.

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

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

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

The **-FilterScript** parameter is positional, and most users will omit it. Most users will also use the **Where** alias, or even the shorter **?** alias. Experienced Windows PowerShell users will also use the **\$\_** variable instead of **\$PSItem**, because only **\$\_** was allowed in version 1.0 and 2.0 of the shell. Both of 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 quotation marks around '**Running**' in these examples are required. Otherwise, the shell will try to run a command called **Running**, which would fail because no such command exists.

## Multiple Criteria

The advanced syntax enables you to specify multiple criteria by using the **-and** and **-or** Boolean or logical operators. For 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 this example, the first comparison checked the **EventID** property, and the second comparison checked the **EntryType** property. The following example is one that many beginning users try. It is incorrect because the second comparison is incomplete.

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

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

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

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

### 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 will produce those values by using a comparison operator, as in the examples that you have seen to this point. However, when a property already contains either True or False, you do not have to explicitly make a comparison. For example, the objects produced by **Get-Process** have a property named **Responding**. This property contains either True or False. This indicates whether the process represented by the object is currently responding to the operating system. To obtain a list of processes that are responding, you could 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. In the second example, there is no comparison at all. The second example works because the **Responding** property already contains True or False.

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

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

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

### No Limits on Accessing Properties

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

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

## Demonstration: Filtering

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

## Demonstration Steps

1. Use basic filtering syntax to display a list of SMB shares that include a dollar sign (\$) in their share name.
2. Use advanced filtering syntax to display a list of physical disks that are in healthy condition.
3. Display a list of disk volumes that are fixed disks and that use the NTFS file system.
4. Using advanced filtering syntax and without using the **\$PSItem** variable, display a list of Windows PowerShell command verbs that begin with the letter **C**.

## Optimizing Filtering Performance

You can have a significant effect on performance based on how you write your commands.

Imagine that you are given a container of plastic blocks. Each block is one of the following colors: red, green, or blue. Each block has a letter of the alphabet printed on it. You are asked to put all the red blocks in alphabetical order. What would you do first?

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

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

```
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 would be faster, because it removes unwanted blocks from the pipeline. The first command sorts all the blocks, and then removes many of them. This means that much of the sorting effort was wasted.

There is a mnemonic used by many Windows PowerShell users to help them remember to do the correct thing when they are optimizing performance. The phrase is *filter left*, and it means that any filtering should occur as far to the left, or as close to the beginning of the command line, as possible.

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

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

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

```
Get-ChildItem -File
```

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

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

Is this the most efficient way to produce a list of services whose names start with svc? No. Here is a better approach:

```
Get-Service -Name svc*
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab C: Filtering Objects

### Scenario

You have to retrieve management information about the computers in your environment. You want the output of your commands to include only specified information and objects.

### Objectives

After completing this lab, students will be able to:

- Filter objects out of the pipeline by using basic and advanced syntax forms

### Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
5. User name: **Administrator**
6. Password: **Pa\$\$w0rd**
7. Domain: **ADATUM**
8. Repeat steps 2 through 4 for 10961B-LON-CL1.
9. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Filtering Objects

#### Scenario

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

Some tasks in this exercise will require you to filter based on date and time information. You should already know a command that can retrieve the current date and time. That command will usually have to be executed enclosed in parentheses so that you can use its result, instead of the command itself, as a comparison value. When you compare dates, any date in the future is considered greater than today's date. Any date in the past is considered less than today's date.

You will also have to calculate free space percentages in this exercise. The mathematical formula to calculate free space percentage is *(Free Space / Size)*.

The main tasks for this exercise are as follows:

1. Display a list of all users in the Users container of Active Directory

2. Create a report that shows Security event log entries having the event ID 4624
3. Display a list of encryption certificates installed on the computer
4. Create a report that shows disk volumes that are running low on space
5. Create a report that displays specified Control Panel items

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

1. Log on to the 10961B-LON-CL1 virtual machine logged in as **Adatum\Administrator**.
2. Using a keyword like **user**, find a command that can list Active Directory® users.
3. View the full Help for the command and identify any mandatory parameters.
4. Display a list of all users in Active Directory.
5. Display a list of all users in the Users container of Active Directory. Use a search base of "**cn=Users,dc=adatum,dc=com**" for this task.

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

1. Display a list of Security event log entries that have the event ID 4624.
2. Display the list again and show only the time written, event ID, and message.
3. Produce the same list in an HTML file named **EventReport.html**.
4. View EventReport.html in Internet Explorer.

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

1. Display a directory listing of all items in the **CERT:** drive. Include subfolders in the list.
2. Display the members of the objects produced by the directory listing.
3. Display the list again and show only certificates that do not have a private key.
4. Display the list again and show only certificates that have a **NotBefore** date that is before today, and a **NotAfter** date that is after today. Include only certificates that do not have a private key.
5. Display the list again and show only the issuer name, **NotAfter** date, and **NotBefore** date for each certificate.

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

1. Display a list of disk volumes.
2. Display a list that shows the members of the objects produced by the previous command.
3. Display a list of volumes that have more than zero bytes of free space.
4. Display a list of volumes that have less than 99 percent free space, and more than zero bytes of free space.
5. Display a list of volumes that have less than 10 percent free space and more than zero bytes of free space. This command may produce no results if no volumes on your computer meet the criteria.

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

1. Using a keyword like **control**, find a command that can display Control Panel items.
2. Display a list of all Control Panel items.
3. Display a list of Control Panel items in the **System and Security** category.

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

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

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

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

## Lesson 5

# Enumerating Objects in the Pipeline

In this lesson, you will learn how to enumerate objects in the pipeline so that you can work with one object at a time.

## Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of enumeration
- Enumerate objects by using basic syntax
- Enumerate objects by using advanced syntax

## The Purpose of Enumeration

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

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

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

- Take a collection of objects...
- ...and execute some action on each one of them, one at a time
- Not necessary when the shell has a command that can perform the action you need
- Useful when an object has a method that will do what you want but the shell does not offer an equivalent command
- The command is **ForEach-Object** (aliases **ForEach** and **%**)

One common scenario that requires enumeration is when you have to execute a method of an object, and no command provides the same functionality as that method. For example, the objects produced by **Get-Object** have a **Kill()** method that stops the process. But you might not ever use that method, because the **Stop-Process** command does the same thing.

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

Enumeration is performed by the **ForEach-Object** command. It has two common aliases: **ForEach** and **%**.

## Basic Enumeration Syntax

Like **Where-Object**, **ForEach-Object** has a basic syntax and an advanced syntax.

In the basic syntax, you can execute a single method, or access a single property, of the objects that were piped into the command. For example:

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

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

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

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



**Note:** You may not discover many scenarios where you have to use enumeration. Windows 8 and Windows Server® 2012, for example, introduced thousands of new Windows PowerShell commands. Many of those new commands perform actions that previously required enumeration.

### Limitations of the Basic Syntax

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

```
Get-Service | Where Name -like 's*' -and Status -eq 'Running'
```

### Demonstration: Basic Enumeration

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

#### Demonstration Steps

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

## Advanced Enumeration Syntax

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

For example, to encrypt a set of files by using the advanced syntax:

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

- Allows you to perform any task by writing commands in a script block
- Use **\$PSItem** or **\$\_** to reference the objects that were piped into the command

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

- Additional parameters allow you to specify actions to take before and after the collection of objects is processed

The **ForEach-Object** command (which has the aliases **ForEach** and **%**) can accept any number of objects from the pipeline. It has a **-Process** parameter that accepts a script block. This script block will execute one time for each object that was piped in. Every time that the script block executes, 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 is executed.

 **Note:** When they are used with the advanced syntax, method names are always followed by opening and closing parentheses, even when the method does not have any input arguments. For methods that do need input arguments, provide them as a comma-separated list inside the parentheses. You may not include a space or other characters between the method name and the opening parentheses.

## Advanced Techniques

In some situations, you may need to repeat a given task for a specified number of times. **ForEach-Object** can be used for that purpose, when you use it at the same time as the range operator. For example:

```
1..100 | ForEach-Object { Get-Random }
```

In this command, the range operator is used to produce integer objects from 1 through 100. The range operator is two periods (..) with no space in between them. Those 100 objects are piped to **ForEach-Object**, forcing its script block to execute 100 times. However, because neither **\$\_** nor **\$PSItem** appear in the script block, the actual integers are not used. Instead, the command **Get-Random** is run 100 times. The integer objects are used only to set the number of times the script block executes.

## Demonstration: Advanced Enumeration

In this demonstration, you will see two ways to use the advanced enumeration syntax to perform tasks against several objects.

### Demonstration Steps

1. Modify all items in the **HKEY\_CURRENT\_USER\Network\** key so that all names are uppercase.
2. Produce a list of process names to a text file that is named **Procs.txt**. Include the current date at the beginning of the file.

**Question:** If you have programming or scripting experience, does **ForEach-Object** look familiar to you?

## Lab D: Enumerating Objects

### Scenario

You are asked to complete several management tasks by using Windows PowerShell. These tasks require you to perform actions on multiple objects.

### Objectives

After completing this lab, students will be able to:

- Enumerate pipeline objects by using basic and advanced syntax forms

### Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name : ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-CL1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Enumerating Objects

#### Scenario

In this exercise, you will write commands that manipulate multiple objects in the pipeline. In some tasks, you have to use enumeration. In other tasks, you will not have to use enumeration. Decide the best approach for each task.

The main tasks for this exercise are as follows:

1. Display a list of key algorithms for all encryption certificates installed on your computer.
2. Use enumeration to produce 100 random numbers
3. Execute a method of a Windows Management Instrumentation (WMI) object
4. To prepare for the next module

► **Task 1: Display a list of key algorithms for all encryption certificates installed on your computer.**

1. Display a directory listing of all items in the **CERT:** drive. Include subfolders in the list.

MCT USE ONLY STUDENT USE PROHIBITED

2. Display the members of the objects produced by the previous command.
3. In the member list, find a method that will retrieve the key algorithm for a certificate.
4. Display a list of key algorithms for each installed certificate.
5. Using **Select-Object** instead of **ForEach-Object**, display a list of all certificates on the computer. Display only columns named **Issuer** and **KeyAlgorithm**. The **KeyAlgorithm** column should be a calculated column that uses the method that you discovered in step 3.

► **Task 2: Use enumeration to produce 100 random numbers**

1. Using a keyword such as **random**, find a command that produces random numbers.
2. View the Help for the command.
3. Run **1..100** to put 100 numeric objects into the pipeline.
4. Run the command again. For each numeric object, produce a random number that uses the numeric object as a seed.

► **Task 3: Execute a method of a Windows Management Instrumentation (WMI) object**

1. Close all applications other than the Windows PowerShell console.
2. Run the command **Get-WmiObject -Class Win32\_OperatingSystem -EnableAllPrivileges**.
3. Display the members of the object produced by the previous command.
4. In the member list, find a method that will restart the computer.
5. Run the command again and use enumeration to execute the method that will restart the computer.

► **Task 4: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have written commands that manipulate multiple objects in the pipeline.

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

## Module Review and Takeaways



**Best Practice:** For best performance, remember to move filtering actions as close to the beginning of the command-line as possible. Sometimes that may mean using a filtering capability of a regular command, instead of using **Where-Object**.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
\$_ does not work.	
\$_ is confusing to read.	

### Review Question(s)

**Question:** \$\_ and \$PSItem were used several times in this module. Why might you decide to use one over the other?

### Real-world Issues and Scenarios

One aspect of Windows PowerShell that can be challenging is that frequently, you can achieve the same result several different ways. Different people may select various techniques based on their experience, but that does not necessarily make one technique better or worse than the others. Consider the following:

```
Get-Service | Select-Object -Property Name  
Gsv | Select Name  
Get-Service | ForEach Name  
Get-Service | % { $_.Name }  
Get-Service | ft name
```

In Windows PowerShell 3.0 or later, these five commands produce the same result: a list of service names. As you explore Windows PowerShell, and especially as you read examples written by other people or provided by your instructor, you should be aware that there is not only one correct way to use Windows PowerShell. Part of using the shell is being able to understand many approaches, arrangements of syntax, and techniques.

# Module 3

## Understanding How the Pipeline Works

### Contents:

Module Overview	03-1
Lesson 1: Passing Data in the Pipeline By Value	03-2
Lesson 2: Passing Data in the Pipeline By Property Name	03-7
Lab: Working with Pipeline Parameter Binding	03-11
Module Review and Takeaways	03-14

## Module Overview

In this module, you will learn how Windows PowerShell™ passes objects from one command to another in the pipeline. The shell has two techniques it can use. Knowing how these techniques work, and which one will be used in a given scenario, lets you construct more useful and complex command lines.



**Additional Reading:** You can read more about Piping and the Pipeline in Windows PowerShell at <http://go.microsoft.com/fwlink/?LinkID=306146>.

### Objectives

After completing this module, students will be able to:

- Pass data by using the **ByValue** technique
- Pass data by using the **ByPropertyName** technique

## Lesson 1

# Passing Data in the Pipeline By Value

In this lesson, you will learn about the first of Windows PowerShell's pipeline techniques. Called **ByValue**, it is the first technique the shell tries to use.

## Lesson Objectives

After completing this lesson, students will be able to:

- Describe how commands receive input
- Identify parameters that can receive input by value
- Pass data in the pipeline by using the **ByValue** technique
- Override the pipeline by using manual parameters
- Specify input by using parenthetical commands instead of the pipeline

## Command Input is Only by Parameter

Windows PowerShell commands can accept input only from one of their parameters. This creates a statement that is an easy statement to read but how the command works may not always obvious when you run a command. For example:

```
Get-Service | Sort-Object -Property Status | Select-Object -Property Name,Status
```

In that example, it seems as if both **Sort-Object** and **Select-Object** are accepting input in some other way. The objects produced by **Get-Service** are somehow being picked up out of the pipeline by **Sort-Object**, and so on. In reality, both **Sort-Object** and **Select-Object** are being given two parameters, not the one parameter that you can see. The other parameter is invisibly used by Windows PowerShell in a process that is known as *pipeline parameter binding*.

Basically, when two commands are connected in the pipeline, pipeline parameter binding has to take the output of the first command and decide what to do with it. The process has to select one of the parameters of the second command to receive those objects. The shell has two techniques that it uses to make that decision. The first technique, and the one that the shell always tries to use first, is called **ByValue**. The second technique is called **ByPropertyName** and is only used when **ByValue** fails.

- Commands accept input only from their parameters
- **Get-Service | Sort-Object -Property Status**
- In the preceding example, only two parameters are specified for **Sort-Object**
  - The one parameter you see, **-Property**
  - Another parameter used invisibly as part of pipeline parameter binding
- Two techniques for pipeline parameter binding
  - **ByValue** is always tried first
  - **ByPropertyName** is tried if **ByValue** fails

## Finding 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:

```
-InputObject <PSObject>
    Specifies the objects to be
    sorted.
    To sort objects, pipe them to
    Sort-Object.
    Required?          false
    Position?         named
    Default value
    Accept pipeline input?   true
    (ByValue)
    Accept wildcard characters? false
```

- Command Help files indicate the parameters that can accept the pipeline input **ByValue**

Required?	false
Position?	named
Default value	
Accept pipeline input?	true (ByValue)
Accept wildcard characters?	false

The **Accept pipeline input?** attribute is **true** because the **-InputObject** parameter accepts pipeline input. Also shown is a list of the techniques the parameter supports. In this case, it supports only the **ByValue** technique.

## Passing Data ByValue

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 will accept a different kind of object.

For example, **Get-Service** can accept pipeline input **ByValue** on both its **-InputObject** and **-Name** parameters. Those parameters each accept a different kind of object. **-InputObject** accepts objects of the type **ServiceController**, and **-Name** accepts objects of the type **String**. Look at this example:

```
'BITS','WinRM' | Get-Service
```

String objects in the pipeline  
"BITS", "WinRM" | Get-Service -Name  
Attach invisibly to the parameter that accepts String objects from the pipeline ByValue

Here, two string objects are being piped into **Get-Service**. They will be attached to the **-Name** parameter, because that parameter accepts that kind of object, **ByValue**, from the pipeline.

The key to predicting what the shell will do with objects in the pipeline is knowing what kind of object is in the pipeline. To do that, you can pipe the object to **Get-Member**. The first line of output will tell you what kind of object the pipeline contained. For example:

```
PS C:\> "BITS", "WinRM" | Get-Member
  TypeName: System.String
  Name      MemberType      Definition
  ----      -----      -----

```

Here, the pipeline contained objects of the type **System.String**. Windows PowerShell frequently abbreviates type names to include only the last portion. In this example, that would be **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 are attached 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, the shell 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, the shell will try to match generic data types. That behavior is why commands like **Sort-Object** and **Select-Object** work. Those commands each have a parameter named **-InputObject** that accepts objects of the type **PSObject** from the pipeline **ByValue**. That is why you can pipe anything to those commands. Their **-InputObject** parameter will receive anything from the pipeline, because it accepts objects of any kind.

## Demonstration: Passing Data ByValue

In this demonstration, you will see how the shell performs pipeline parameter binding **ByValue**.

### Demonstration Steps

1. Look at this example 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 what parameter of the second command in step 1 will receive the output of the first command.

## Manual Parameters 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 the shell to select another parameter for pipeline parameter binding. For example:

```
Get-Content Names.txt | Get-Service -  
Name BITS
```

Imagine that **Names.txt** contains a list of one service name per line. The **Get-Content** command reads those into the shell, putting each name into the pipeline as a string object.

- The shell first looks to see what parameter it wants to use for pipeline parameter binding...
- ...but if that parameter is specified manually, binding stops.

```
Get-Content Names.txt | Get-Service -Name BITS
```

- The manual use of **-Name** overrides the pipeline input. The pipeline input is not used. You actually see an error.

Windows PowerShell would typically attach those strings to the **-Name** parameter of **Get-Service**. However, in this example, the **-Name** parameter was already used manually. That puts a stop to pipeline parameter binding. The shell will not look for another possible parameter to bind the input to. The parameter it wanted to use is taken, so that the process is over.

Frequently, you will receive an error. For example:

```
PS C:\> "BITS","WinRM" | Get-Service -Name BITS
Get-Service : 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 that **the command does not take pipeline input**. However, the command usually does. In this example, you have disabled the command's ability to accept the pipeline input because you manually specified the parameter that the shell wanted to use.

## Demonstration: Overriding the Pipeline

In this demonstration, you will see examples of what happens when you manually specify a parameter that the shell would usually have used in pipeline parameter binding.

### Demonstration Steps

1. Pipe two strings to display a list of two services.
2. Pipe two strings to display a list of services, but manually specify a service name.

## Parenthetical Commands Instead of the Pipeline

Sometimes you want to provide input to a command when the command's parameters do not accept the appropriate input from the pipeline. In other scenarios, you may have to provide input to a command but cannot discover how the command works with pipeline input. In those cases, it may be better to rely on a parenthetical command instead of relying on the pipeline.

For example, the following command works to display a list of services:

```
Get-Content ServiceNames.txt | Get-Service
```

- Any command or commands in parentheses will be run first
- The results will be inserted in place of the parenthetical command
- Works with any parameter, provided that the command produces the kind of object the parameter expects

```
Get-Process -Name (Get-Content
Names.txt)
```

However, the following command does not work, because the second command was not designed to accept the pipeline input provided:

```
Get-Content ProcessNames.txt | Get-Process
```

In the second example, a parenthetical command can be used:

```
Get-Process -Name (Get-Content ProcessNames.txt)
```

A *parenthetical command* is a command that is enclosed in parentheses. Just as in math, parentheses tell the shell to “do this first.” The parenthetical command runs, and the results of the command are inserted in its place. In this example, the contents of the ProcessNames.txt file would be inserted as input to the **-Name** parameter.

Parenthetical commands do not rely on pipeline parameter binding. They work with any parameter as long as the parenthetical command produces the kind of object that the parameter expects.

## Demonstration: Parenthetical Commands

In this demonstration, you will see examples of using parenthetical commands.

### Demonstration Steps

1. Create a text file that contains **LON-CL1** on one line, and **localhost** on the other.
2. Verify the contents of the text file.
3. Use a parenthetical expression to display a list of services that are running on LON-CL1 and localhost.

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

## Lesson 2

# Passing Data in the Pipeline By Property Name

In this lesson, you will learn about the **ByPropertyName** technique of passing data in the pipeline. This is the second technique that Windows PowerShell tries to use.

### Lesson Objectives

After completing this lesson, students will be able to:

- Display a list of object properties
- Identify parameters that can accept input by property name
- Pass data by using the **ByPropertyName** technique
- Expand object properties into simple values

### Changing to ByPropertyName

If Windows PowerShell is unable to bind pipeline input by using the **ByValue** technique, it tries to use the **ByPropertyName** technique.

For example:

```
Get-Service | Stop-Process
```

The first command puts objects of the type **ServiceController** 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.

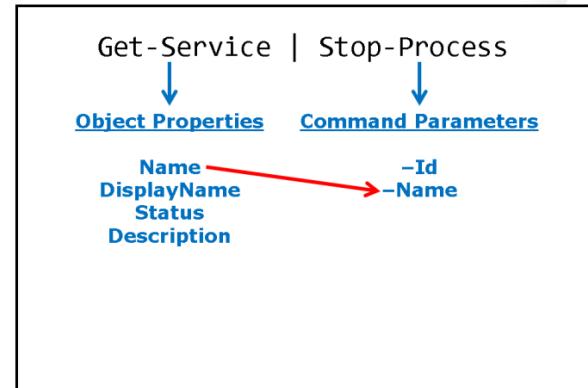
The shell changes to the **ByPropertyName** technique. To predict what it will do, you must review the properties of the objects produced by the first command. In this example, you would run:

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

You would also have to make a list of parameters, of the second command, that can accept pipeline input by using **ByPropertyName**. To make that list, view the Help for the second command:

```
Help Stop-Process -ShowWindow
```

In this example, the **Stop-Process** command has more than one parameter that accepts pipeline input by using **ByPropertyName**. Those parameters are **-Name** and **-Id**. The objects produced by **Get-Service** do not have an ID property. So, the **-Id** parameter is not considered. The objects produced by **Get-Service** have a **Name** property. Therefore, the contents of the **Name** property are attached to the **-Name** parameter.





**Note:** The **ByPropertyName** technique is very simplistic. Because the property **Name** and the parameter **-Name** are spelled the same, and because the parameter was programmed to accept input in this manner, they connect to one another.

## Finding ByPropertyName Parameters

To see what parameters accept pipeline input by using **ByPropertyName**, examine the full Help for the command.

It is possible for a parameter to accept pipeline input by using both **ByValue** and **ByPropertyName**. Remember that the shell will always try **ByValue** first, and will use **ByPropertyName** only if **ByValue** fails.

- The full Help for a command lists the parameters that accept pipeline input by using **ByPropertyName**.

Required?	false
Position?	named
Default value	Local computer
Accept pipeline input?	true ( <b>ByPropertyName</b> )
Accept wildcard characters?	false

## Demonstration: Passing Data ByPropertyName

In this demonstration, you will see how **ByPropertyName** can be used to create new user accounts.

### Demonstration Steps

- Display the contents of a comma-separated value (CSV) file that contains new user information.
- View the Help for **New-ADUser** and locate parameters that can accept pipeline input.
- Using **-WhatIf**, create new users by using the information in the CSV file.
- View a second CSV file that represents a more real-world scenario.
- Run a command that imports the second CSV file, modifies the objects, and creates new Active Directory® users.
- Verify that the new users were created.

## Expanding Property Values

In the previous lesson, you learned how parenthetical commands can be used to provide parameter input without using the pipeline. In some cases, you may have to manipulate the objects produced by a parenthetical command so that the command's output is of the type the parameter requires.

For example, suppose that you wanted to list all the processes running on every computer in the domain. For this example, imagine that you have a very small lab domain that contains just a few computers. You can get a list of every computer in the domain by running this command:

- The **-ExpandProperty** parameter of **Select-Object** expands, or extracts, the contents of a single property
- Instead of returning an object with many properties, the command returns a simpler value
- This technique can be used in parenthetical commands that need to provide a simpler value to a parameter

```
Get-ADComputer -filter *
```

However, that command produces objects of the type **ADComputer**. You could not use those objects directly in a parenthetical command such as this one:

```
Get-Process -ComputerName (Get-ADComputer -filter *)
```

The **-ComputerName** parameter expects objects of the type **String**. But that is not what the parenthetical command is producing. Basically, the **-ComputerName** parameter just wants a computer name. However, you are giving it an object that contains a name, an operating system version, and several other properties.

Would this work?

```
Get-Process -ComputerName (Get-ADComputer -filter * | Select-Object -Property Name)
```

That command selects only the **Name** property. But that property is still a member of a whole **ADComputer** object. It is the **Name** property of an object. Although the **Name** property contains a string, it is not itself a string. The **-ComputerName** parameter expects a string, not an object having a property.

This command will achieve the goal:

```
Get-Process -ComputerName (Get-ADComputer -filter * | Select-Object -ExpandProperty Name)
```

The **-ExpandProperty** parameter accepts one, and only one, property name. When you use that parameter, only the contents of the specified property are produced by **Select-Object**. Some people refer to this as *extracting the property contents*. The official description of the feature is *expanding the property contents*.

In the preceding command, the result of the parenthetical command is a collection of strings, and that is what the **-ComputerName** parameter expects. The command will work correctly, although of course it may produce an error if one or more of the computers cannot be reached on the network.

## Demonstration: Expanding Property Values

In this demonstration, you will see how to use parameter expansion (extraction) to provide input from a parenthetical command.

### Demonstration Steps

1. Open the Windows PowerShell Integrated Scripting Environment (ISE).
2. Run a command that will list all computers in the domain.
3. Run a command that uses a parenthetical command to display a list of services from every computer in the domain.
4. Run a command that shows the kind of object produced when you retrieve every computer from the domain.
5. Review the help for **Get-Service** to see what kind of object its **-ComputerName** parameter expects.
6. Run a command that selects only the **Name** property of every computer in the domain.
7. Run a command that shows the kind of object produced by the previous command.

8. Run a command that extracts the contents of the **Name** property of every computer in the domain.
9. Run a command that shows the kind of object produced by the previous command.
10. Modify the command in step 3 to use the command in step 8 as the parenthetical command.
11. Run the command created in step 10.

**Question:** Can correct use of pipeline parameter binding reduce the need to use **ForEach-Object**?

# Lab: Working with Pipeline Parameter Binding

## Scenario

You are creating and troubleshooting Windows PowerShell commands. You have to predict and control how the shell will pass data from one command to another so that the commands run correctly.

## Objectives

After completing this lab, students will be able to:

- Predict the behavior of commands in the pipeline
- Write commands that comply with specified pipeline behavior

## Lab Setup

Estimated Time: 45 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. This lab can be performed with the use of the 10961B-LON\_CL1 virtual machine.

## Exercise 1: Predicting Pipeline Behavior

### Scenario

You have to review several Windows PowerShell commands and determine whether they will work. Some commands use pipeline input, but other commands do not. Without running the commands in their entirety, you have to decide whether they will achieve the stated goal.

You also have to write several Windows PowerShell commands that will achieve stated goals. You must not run these commands in the shell. Instead, write them on paper.

The main tasks for this exercise are as follows:

1. Review existing commands

2. Write new commands that perform specified tasks
3. To prepare for the next module

► Task 1: Review existing commands

For these tasks, you may run individual commands and **Get-Member** to see what kinds of objects the commands produce. You may also view the Help for any of these commands.

However, do not run the whole command shown. If you do run the whole command, it may produce an error. The error does not mean the command is written incorrectly.

1. This command is intended to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Get-Service -Name *
```

Will the command achieve the goal?

2. This command is intended to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-Service -Name *
```

Will the command achieve the goal?

3. This command is intended to query an object from every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-WmiObject -Class Win32_BIOS
```

Will the command achieve the goal?

4. The file **Names.txt** lists one computer name per line.

5. This command is intended to list the services that are running on every computer that is listed in **Names.txt**.

```
Get-Content Names.txt | Get-Service
```

Will the command achieve the goal?

6. The file **Names.txt** lists one computer name per line.

This command is intended to list the services that are running on every computer that is listed in **Names.txt**.

```
Get-Service -ComputerName (Get-Content Names.txt)
```

Will the command achieve the goal?

7. This command is intended to list the services that are running on every computer in the domain:

```
Get-Service -ComputerName (Get-ADComputer -Filter *)
```

Will the command achieve the goal?

8. This command is intended to list the Security event log entries from every computer in the domain:

```
Get-EventLog -LogName Security -ComputerName (Get-ADComputer -Filter * | Select -Expand Name)
```

Will the command achieve the goal?

► **Task 2: Write new commands that perform specified tasks**

In each of these tasks, you are asked to write a command that achieves a specified goal. *Do not run these commands.* Write them on paper.

You may run individual commands and pipe their output to **Get-Member** to see what objects those commands produce. You may also read the Help for any command.

1. Write a command that uses **Get-EventLog** to display the most recent 50 System event log entries from each computer in the domain.
2. You have a text file that is named **Names.txt** that contains one computer name per line. Write a command that uses **Restart-Computer** to restart each computer that is listed in the file. Do not use a parenthetical command.
3. You have a file that is named **Names.txt** that contains one computer name per line. Write a command that uses **Test-Connection** to test the connectivity to each computer that is listed in the file.
4. Write a command that uses **Set-Service** to set the start type of the **WinRM** service to **Auto** on every computer in the domain. Do not use a parenthetical command.

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have reviewed and written several Windows PowerShell™ commands.

**Question:** Why do some commands accept pipeline input for a parameter such as – **ComputerName**, but other commands do not?

**Question:** Do you ever have to rely on pipeline input? Could you just rely on parenthetical commands?

## Module Review and Takeaways



**Best Practice:** It is easy to start using Windows PowerShell and not think about what the shell is doing for you. Always take a moment to examine each command that you write, and think about what the shell will do. Think about what objects will be produced by each command, and how those will be passed to the next command.

### Review Question(s)

**Question:** Because pipeline input binding is handled invisibly by the shell, it can be difficult to troubleshoot. Are there any tools that can help you troubleshoot pipeline input?

### Real-world Issues and Scenarios

Sometimes, command authors do not realize how useful and important pipeline input can be, and they do not create their parameters to accept pipeline input. All that you can do in those cases is submit a request to the command author to support pipeline input in a future release.

# Module 4

## Using PSProviders and PSDrives

### Contents:

Module Overview	04-1
Lesson 1: Using PSProviders	04-2
Lesson 2: Using PSDrives	04-5
Lab: Using PSProviders and PSDrives	04-9
Module Review and Takeaways	04-12

## Module Overview

In this module, you will learn to work with **PSProviders** and **PSDrives**. A **PSProvider** is basically a Windows PowerShell™ adapter that makes some form of storage resemble a disk drive. A **PSDrive** is an actual connection to a form of storage. These two technologies let you work with many forms of storage by using the same commands and techniques that you use to manage the file system.



**Additional Reading:** You can read more about Managing Windows PowerShell Drives at <http://go.microsoft.com/fwlink/?LinkID=306147>.

### Objectives

After completing this module, students will be able to:

- Explain the purpose and use of PSProviders
- Explain the purpose and use of PSDrives

## Lesson 1

# Using PSProviders

In this lesson, you will learn about **PSProviders**, which are the adapters that connect Windows PowerShell to data stores.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of **PSProviders**
- Display **PSProvider** Help files
- Compare different **PSProvider** capabilities

### What are Providers?

As you learned in the Module Overview, a **PSProvider**, or just *provider*, is an adapter that makes an external data store resemble a disk drive within Windows PowerShell. Because most administrators are already familiar with command-line management of disk drives, **PSProviders** help those administrators manage other forms of data storage by using familiar commands.

A provider presents data as a hierarchical store. Items like folders can have subitems like subfolders. Items can also have properties, and by using some providers, you can manipulate both items and item properties by using a specific set of commands.

Managing a technology by using a provider is somewhat more difficult than managing by using commands. Commands are typically specific, and the command name describes what the command does. For example, in Microsoft® Exchange Server 2007 and newer versions, the **Get-Mailbox** command retrieves mailbox objects. If that technology used a **PSProvider** instead, you might have to run a command like **Get-ChildItem EXCHANGE:\Server2\MailStore\Mailboxes** instead.

- Adapts data stores to look like disk drives inside the shell
- Allows management by using familiar file system management commands
- Good solution for dynamic or extensible technologies where all manageable components cannot be known in advance
- More complex than managing by using commands

 **Note:** That command is just a theoretical example. Exchange Server is not managed by using a **PSProvider**. However, you can see that managing by using a **PSProvider** looks somewhat more complex.

The advantage of a **PSProvider** is that it is dynamic. For example, Microsoft cannot know in advance what disk drives, folders, and files that you will install and create on a computer. The **FileSystem** **PSProvider** can dynamically adapt to whatever each computer contains. Microsoft Internet Information Services (IIS) is also managed, in part, by using a **PSProvider**. That provider can adapt to whatever Microsoft and third-party add-ins are installed in IIS. Microsoft could not easily write commands to manage everything in IIS,

because new add-ins are created constantly. So even though management by using a provider is more complex, it is a better strategy for dealing with dynamic and extensible technologies.

## Accessing Provider Help

You can display a list of available providers by running **Get-PSProvider**. Be aware that providers can be added into the shell when you load modules. For example, running **Import-Module ActiveDirectory** will load the Active Directory® module. That module includes a PSProvider.

When you know the name of a provider, you can view its Help, if Help is provided. For example, run **Help FileSystem** to display Help on the **FileSystem** PSProvider. Provider Help files frequently contain descriptions and examples that are specific to the data store that the provider connects to.

The commands that are used to work with providers use the generic nouns **Item** and **ItemProperty**. Because the commands are designed to work with any provider, the examples in the commands' Help may not include examples for every scenario. Provider Help is intended to supplement command Help with more specific descriptions and examples.

- Run **Get-PSProvider** for a list of providers
- Run **Help <provider-name>** for provider-specific Help
- Provider-specific Help can offer better descriptions and examples than the Help for the generic commands that you use when working with providers and drives

## Different Provider Capabilities

Running **Get-PSProvider** also lists the capabilities of each provider that is loaded into the shell. Because each provider connects to a different underlying technology, the capabilities of each provider will be different.

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 a **-UseTransaction** parameter. However, only the **Registry** PSProvider supports the **Transactions** capability. If you try to use the parameter in any other provider, you will receive an error message.

Some major capabilities include the following:

- Each provider can support one or more of several different capabilities
- If a provider does not support a capability, you cannot use the corresponding parameters of commands that work in a provider
- For example, only the **Registry** provider supports the **Transactions** capability, so only that provider can accept the **-UseTransaction** parameter

- **ShouldProcess**, for providers that can support the **-WhatIf** and **-Confirm** parameters
- **Filter**, for providers that support filtering
- **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 so that you will not encounter unexpected errors by trying to use unsupported capabilities.

## Demonstration: Working with Providers

In this demonstration, you will see how to work with providers.

### Demonstration Steps

1. Display a list of providers. Notice the capabilities listed for each one.
2. Load the Active Directory module.
3. Display a list of providers. Notice the new provider added by the Active Directory module.
4. Display Help for a provider.

**Question:** What other kinds of PSProviders might exist as add-ins to the shell?

## Lesson 2

# Using PSDrives

In this lesson, you will learn how to work with PSDrives. A *PSDrive* represents a specific form of storage that is connected to the shell by using a PSProvider.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose and use of PSDrives
- Manage the file system
- Navigate a PSDrive
- Create and modify items and item properties

### What Are Drives?

A PSDrive, or *drive*, is a connection to a data store. Each PSDrive uses a single PSProvider to connect to a data store. The PSDrive has all the capabilities of the PSProvider that is used to make the connection.

Drives in Windows PowerShell are identified by names, not only by single letters, although a name can consist of just one letter. For example, the drive HKCU connects to the **HKEY\_CURRENT\_USER** registry hive. Single-letter drive names typically connect to a **FileSystem** drive. For example, drive C connects to the physical drive C of your computer.

- A drive represents a connected data store
- Drives use a PSProvider to connect to the store
- Drives have a name like C or Alias
- Run **Get-PSDrive** for a list
- Run **New-PSDrive** to map a new drive
- Shell always starts with the same drives mapped

To create a new connection, use the **New-PSDrive** command. You must specify a unique drive name, the root location for the new drive, and the PSProvider that will be used to make the connection. Depending on the capabilities of the PSProvider, you may also specify alternative credentials and other options.

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

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

You can see a list of drives by running **Get-PSDrive**.

MCT USE ONLY. STUDENT USE PROHIBITED



**Note:** Drive names do not include the colon. Drive name examples include Variable and Alias. However, when you want to refer to a drive as a path, include the colon. For example, **Variable:** refers to the drive Variable, just as **C:** refers to drive C. Commands like **New-PSDrive** require a drive name. Do not include a colon in the drive name when using those commands.

## Working with the File System

Many administrators already know commands to manage a file system. Common commands include **Dir**, **Move**, **Ren**, **RmDir**, **Del**, **Copy**, **MkDir**, and **Cd**. In Windows PowerShell, these common commands are provided as aliases to Windows PowerShell cmdlets.

Because the cmdlets are not exact duplicates of the original commands, their syntax is different. For example, instead of running **Dir /s** to obtain a directory listing that includes subdirectories, you run **Get-ChildItem -Recurse** to obtain a directory listing that includes subfolder recursion. The parameters are the same whether you decide to use the cmdlet name or the alias. That means that you can run **Dir -Recurse**, but not **Dir /s**.

Common File System Command	Windows PowerShell cmdlet
Dir	Get-ChildItem
Move	Move-Item
Ren	Rename-Item
Del, RmDir	Remove-Item
Copy	Copy-Item
MkDir	New-Item
Cd	Set-Location Get-Location Get-ItemProperty Set-Item Set-ItemProperty



**Note:** Because Windows PowerShell accepts a forward slash or backward slash as a path separator, **Dir /s** is interpreted by the shell as "display a directory listing of the folder named s." Unless that folder exists, you will see an error when you run that command.

Notice that the Windows PowerShell cmdlet names all use generic nouns like **Item** and **ItemProperty**. That is because the cmdlets are intended to work with any PSDrive connected by using any PSProvider. In the file system, an item might be a file or a folder. In the registry, an item might be a key or a setting. In the certificate store, an item might be a folder or a certificate. Instead of having separate commands for all these items, Microsoft chose to create a single set of commands that uses generic nouns.

## How the File System is Like Other Data Stores

Most administrators can answer the question, "What command would you run to delete a file that is named C:\Test.txt?" The answer is **Del C:\Test.txt** or **Remove-Item C:\Test.txt**.

Windows PowerShell stores its own variables in a drive named Variable. With that piece of information, what command would you run to delete a variable named X? The answer is **Del Variable:X**. Because the variable storage is made to resemble a disk drive, you manage it by using the same commands you would use with a disk drive.



**Note:** You can even delete some of Windows PowerShell's built-in variables. That action is not permanent. Every time you start a new Windows PowerShell session, that session creates the same default set of variables, with the same default values. So if you accidentally delete a built-in variable, just close the session and open a new one.

## Specifying Paths

When you use commands that have the **Item**, **ChildItem**, and **ItemProperty** nouns, you will typically specify a path to tell the command what item or items that you want to manipulate. Most of these commands have two parameters for paths:

- **-Path** typically interprets the asterisk (\*) and the question mark (?) as wildcard characters. In other words, the path **\*.txt** refers to "all files ending in .txt." This approach works correctly in the file system, because the file system does not allow item names to contain the characters \* or ?.
- **-LiteralPath** treats all characters as literals, and does not interpret any characters as a wildcard. The literal path **\*.txt** means "the item named \*.txt." This approach is useful in drives where \* and ? are allowed in item names, such as in the registry.

- Most commands that use the noun **Item**, **ChildItem**, or **ItemProperty** have parameters that accept a path
  - **-Path** parameters typically interpret \* and ? as wildcards
  - **-LiteralPath** parameters treat all characters as literal characters

## Working with Items and Item Properties

PSDrives contain items, and items have item properties.

Run **Get-Command -Noun Item,ChildItem** to see a list of commands that work with items. *Items* are things such as files, folders, registry keys, certificates, and variables.

Run **Get-Command -Noun ItemProperty** to see a list of commands that work with item properties. *Item properties* are things such as registry settings. Item properties typically have a name and a value.

The items in different PSDrives behave differently.

For example, in a **FileSystem** drive, a file is an item. The file has content, and with simple text files, you can display that content by using **Get-Content**, or change the content by using **Set-Content**. Files also have properties, such as their name and last write time.

In the drive HKCU, items are registry keys. These registry keys can contain other keys, and can have their own item properties. For example, **HKCU:\Volatile Environment** has item properties named **LOGONSERVER**, **USERDOMAIN**, **HOMEPATH**, and so on. That key also contains other registry keys that have their own item properties. Use **Get-ItemProperty** and **Set-ItemProperty** to work with item properties.

- PSDrives contain items, and items can have item properties
- Run **Get-Command -Noun Item,ChildItem** for a list of item commands
- Run **Get-Command -Noun ItemProperty** for a list of item property commands
- Each PSDrive will have a different definition of what an item is: file, folder, registry key, and so on

## Demonstration: Working with Drives and Items

In this demonstration, you will see how to work with PSDrives, items, and item properties.

**Demonstration Steps**

1. Change to the **C:\** location.
2. Map a new, temporary drive named **WINDIR** to the C:\Windows folder.
3. Display a directory listing of drive WINDIR.
4. Create a new registry key named **HKCU:\Software\Classroom**.
5. Add an item property named **Test**, having the value **1**, to HKCU:\Software\Classroom.
6. Display a directory listing of HKCU:\Software\Classroom.

**Question:** If **Get-Content** displays the contents of a text file on the file system, how could you display the contents of a built-in function like **Help**?

# Lab: Using PSProviders and PSDrives

## Scenario

You have to reconfigure several settings in your environment. These settings are available by using a PSProvider.

## Objectives

After completing this lab, students will be able to:

- Create new items in a PSDrive
- Create new PSDrive mappings
- Create new item properties in a PSDrive
- Modify items and properties in a PSDrive

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The exercise steps should be performed using the 10961B-LON-CL1 virtual machine throughout this lab.

## Exercise 1: Create a New Folder

### Scenario

In this exercise, you will create a new folder on the file system. You may not use the **Mkdir** command or any of its aliases.

The main tasks for this exercise are as follows:

1. Create a new folder

► **Task 1: Create a new folder**

1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
2. Read the full Help for the **New-Item** command.
3. Use **New-Item** to create a new folder (directory) named **C:\ScriptOutput**.

**Results:** After completing this exercise, you will have created a new folder on the file system.

## Exercise 2: Create a New PSDrive

### Scenario

In this exercise, you will create a new, temporary PSDrive.

The main tasks for this exercise are as follows:

1. Create a new PSDrive

► **Task 1: Create a new PSDrive**

1. Read the full Help for the **New-PSDrive** command.
2. Create a new PSDrive named **Output**: and map it to the **C:\ScriptOutput** folder.

**Results:** After completing this exercise, you will have created a new, temporary PSDrive.

## Exercise 3: Create a New Registry Key

### Scenario

In this exercise, you will create a new registry key. The key will be used by scripts that you create to store configuration information.

The main tasks for this exercise are as follows:

1. Create the registry key

► **Task 1: Create the registry key**

1. Read the full Help for the **New-Item** command.
2. Create a registry key named **Scripts** in **HKEY\_CURRENT\_USER\Software**.

**Results:** After completing this exercise, you will have created a new registry key.

## Exercise 4: Create a Registry Setting

### Scenario

The registry key **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run** lists programs that run every time that the operating system starts. Each program is a property of the key. The name of the property is the program name, and the value of the property is the program path. You will add a new program to the list.

The main tasks for this exercise are as follows:

1. Create a new registry setting

► **Task 1: Create a new registry setting**

1. Read the full Help for the **New-ItemProperty** command.
2. Create a new item property under HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run. Give the new item property the following attributes:
  - Name: **Windows PowerShell**
  - Value: **C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe**

**Results:** After completing this exercise, you will have added a new program to the autorun list.

## Exercise 5: Modify a WS-Management Setting

### Scenario

Many settings related to Windows PowerShell™ remoting and the WS-MAN protocol, are accessed by using the PSDrive **WSMAN**: In this exercise, you will modify the maximum number of concurrent incoming connections.

The main tasks for this exercise are as follows:

1. Modify a WS-MAN setting
2. To prepare for the next module

► **Task 1: Modify a WS-MAN setting**

1. Read the full Help for the **Set-Item** command.
2. Modify the **WSMAN:\localhost\Service\MaxConnections** setting so that it has the value **250**.

► **Task 2: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have modified the maximum number of concurrent connections for Windows PowerShell remoting.

**Question:** Of the PSProviders included with Windows PowerShell, which support the use of alternative credentials?

**Question:** Windows PowerShell 3.0 and later can make one kind of PSDrive visible in File Explorer. What kind of drive is that, and how do you make it visible?

## Module Review and Takeaways

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
A PSDrive that was present in the shell is no longer present.	

### Review Question(s)

**Question:** What is the advantage of managing something by using a PSProvider instead of commands?

### Real-world Issues and Scenarios

A parameter of **Get-ChildItem** does not work with a particular **PSDrive**. For example, **-Filter** does not work when listing information in a registry drive. This is a known issue, and happens because each PSProvider has different capabilities. The Registry PSProvider does not support **-Filter**.

# Module 5

## Formatting Output

### Contents:

Module Overview	05-1
Lesson 1: Using Basic Formatting	05-2
Lesson 2: Using Advanced Formatting	05-5
Lesson 3: Redirecting Formatted Output	05-8
Lab: Formatting Output	05-11
Module Review and Takeaways	05-15

## Module Overview

In this module, you will learn how to format the output of commands. Formatting enables you to produce more professional-looking output, and to produce output that is better customized for your specific needs.

 **Additional Reading:** This module mentions the ability to create custom formatting views in Windows PowerShell™. You can read more about that ability at <http://go.microsoft.com/fwlink/?LinkID=306148>.

 **Additional Reading:** Although you can create simple management reports by using Windows PowerShell™ formatting, more complex reports can be created by using Microsoft® SQL Server® Reporting Services (SSRS). For more information, read “Making Historical and Trend Reports in Windows PowerShell,” a free paper available at [www.PowerShellBooks.com](http://www.PowerShellBooks.com).

### Objectives

After completing this module, students will be able to:

- Format command output by using basic formatting commands
- Format command output by using advanced formatting options
- Redirect formatted output

## Lesson 1

# Using Basic Formatting

In this lesson, you will learn about the three basic formatting commands in Windows PowerShell.

### Lesson Objectives

After completing this lesson, students will be able to:

- Describe the default formatting process
- Display output in wide lists
- Display output in lists
- Display output in tables

### Default Formatting

To this point, all the commands that you have run in Windows PowerShell have used the shell's default formatting system to produce their results on the screen. This formatting system uses the following rules:

1. Find the type name of the first object in the pipeline.
2. Determine whether that object type has a default formatting view. These views are defined in XML files. Several view definition files are included with Windows PowerShell, and third parties can provide additional view definitions. If the object type has a defined view, the shell will use it.
3. If the object type does not have a defined view, decide whether the object type has a default display property set. If it does, the properties in that set will be displayed. If there are five or more properties, they will be displayed as a list. Otherwise, they will be displayed as a table.
4. If the object type does not have a default display property set, all the properties will be displayed. If there are more than four properties, they will be displayed as a list. Otherwise, they will be displayed as a table.

- Decides how to display output based upon the object type of the first object in the pipeline
- Uses a set of XML view definitions and XML type extensions to decide what to display, and how to display it
- You can manipulate these defaults by providing additional XML files, although creating those files is beyond the scope of this course

You can define your own views and property sets to change the default appearance of a specified object type. Doing this is beyond the scope of this course. However, you can read the Help for **Update-FormatData** and **Update-TypeData** for more information.

## Wide Lists

You can pipe objects to the **Format-Wide** command to produce a wide list. The command defaults to a two-column display of the objects' **Name** properties (because most objects have a **Name** property). The command has the alias **FW**.

You can modify the output by using parameters:

- Use **-Column** to specify the number of columns to display, or use **-AutoSize** to let the command calculate the maximum number of columns that will fit.
- Use **-Property** to specify a single property to display other than **Name**.

```
Get-Service | Format-Wide
Get-Process | Format-Wide -Property ID
Get-Process | Format-Wide -Col 5
Get-Process | FW -AutoSize
```

## Lists

You can pipe objects to the **Format-List** command to display objects' properties in a list format. If you do not specify properties to display, and the object has a set that is defined, the objects' default display property set will be shown. Otherwise, all the objects' properties will be shown. **Format-List** has the alias **FL**.

You can customize the output by using parameters. Use **-Property** to specify a comma-separated list of property names to display. Or, specify the asterisk (\*) wildcard to display all properties. This is a useful technique for quickly seeing all an object's properties and their values.

**Format-List** does not contain other parameters that you will use frequently. Read the Help for the command to learn about its other parameters.

```
Get-Service | Format-List -Property *
Get-Process | FL -Prop Name, ID
Get-Service | FL Name, Status, DisplayName
```

## Tables

You can pipe objects to the **Format-Table** command to display object properties in a columnar format. **Format-Table** has the alias **FT**.

By default, the command will display the properties in the objects' default display property set, if one is defined. The command will always try to fill the width of the screen, and will add space between columns to fill the screen.

You can customize the output by using parameters:

- The **-Property** parameter accepts a comma-

```
Get-Process | Format-Table -Property ID, Name
Get-Process | FT *
Get-Service | FT Name, Status -AutoSize -Wrap
```

separated list of property names. You can use the wildcard \* to display all properties. However, for objects having lots of properties, the command may eliminate columns that do not fit.

- The **-AutoSize** parameter tries to size each column to hold its widest item, and eliminates additional space between columns.
- The **-Wrap** parameter enables column content to wrap across several lines. This prevents data from being truncated.

## Demonstration: Basic Formatting

In this demonstration, you will see how to use the basic features of the three main formatting commands.

### Demonstration Steps

1. Display a list of process ID numbers in a two-column wide list.
2. Display a list of process ID numbers in a five-column wide list.
3. Display a list of process names in as many columns as will fit on the screen.
4. Display a list that shows only service names and statuses.
5. Display a list that shows all properties of every process.
6. Display a table that shows the most recent 50 entries from the Security event log. Display only the time written and event ID for each entry. Do not allow additional space between table columns.

**Question:** Look at the full Help for one of the format commands. What parameter do they all use to accept input from the pipeline?

## Lesson 2

# Using Advanced Formatting

In this lesson, you will learn to use some of Windows PowerShell's advanced formatting options and techniques.

### Lesson Objectives

After completing this lesson, students will be able to:

- Create custom list entries and columns
- Sort and group table output
- Explain the differences between **Select-Object** and format cmdlets

### Custom List Entries and Columns

The format commands support the same calculated property syntax as **Select-Object**. In addition to the **n** (or **name**, or **I**, or **label**) and **e** (or **expression**) keys, you can specify other keys to control the appearance of the output:

- For lists and tables, **formatString** specifies a formatting string. For example, **N2** specifies a number that has two decimal places.
- For tables, **align** specifies the table alignment. **Left**, **Right**, and **Center** are valid values.
- For tables, **width** specifies maximum column width. The table column may be narrower if no data item requires the specified width. Longer data may be truncated.

For example:

- Extension of the calculated property syntax used by **Select-Object**:

```
Get-Process |  
Format-Table -Property Name, ID,@{n='VM(MB)';  
e={$PSItem.VM / 1MB};  
formatString='N2';  
align='right'} -  
AutoSize
```

```
Get-Process |  
Format-Table -Property Name, ID,@{n='VM(MB)';  
e={$PSItem.VM / 1MB};  
formatString='N2';  
align='right'} -AutoSize
```

That command produces output as follows:

Name	Id	VM(MB)
---	--	-----
conhost	3948	59.28
csrss	584	48.38
csrss	648	48.27
dllhost	2084	55.21
dwm	964	375.39

 **Note:** Remember that **\$PSItem** was introduced in Windows PowerShell 3.0. Earlier versions used **\$\_**, and you can continue to use **\$\_** in all versions if you prefer.

## Demonstration: Custom Table Columns

In this demonstration, you will see how to create custom table columns by using **Format-Table**.

### Demonstration Steps

1. Display a list of local hard disks that includes free space in gigabytes, size in gigabytes, and description. Limit the description to five characters and do not include additional space between columns.
2. Display the same list again, but allow additional space between columns.

## Sorted and Grouped Tables

**Format-Table** has a **-GroupBy** parameter that accepts a property name. Every time the value of that property changes, the command produces a new set of table headers. For example:

```
Get-Service | Format-Table -GroupBy Status
```

So that the group values change only once per unique value, you will typically sort the objects on that property before formatting them:

```
Get-Service | Sort Status | Format-Table -GroupBy Status
```

- **-GroupBy** parameter of format commands accepts a property name
- Produces a new set of headers each time that property's value changes
- For best results, first sort the objects on the same property

```
Get-Service | Sort Status | Format-Table -GroupBy Status
```

**Format-Wide** and **Format-List** also support a **-GroupBy** parameter.

## Demonstration: Grouped Tables

In this demonstration, you will see how to display grouped table output.

### Demonstration Steps

1. Display a list of services, grouped by status. Do not sort the objects.
2. Display a list of services, sorted by status and grouped by status.

## Select or Format?

There is overlapping functionality between the **Select-Object** command and the formatting commands. Both accept a property list that controls the properties displayed. Both can accept custom, or *calculated*, properties.

Use **Select-Object** when you have to pipe objects to any command other than a formatting command. For example, if you have to sort, filter, enumerate, export, or convert objects, use **Select-Object** to select properties and create calculated properties.

Use formatting commands only when you are finished working with the objects and are ready to display them on the screen or in a plain text file.

You will learn more about the differences between formatting commands and **Select-Object** in the next lesson.

- **Select-Object** and the format commands, especially **Format-Table**, have overlapping functionality
- Use **Select** if you need to pipe objects to another command to sort, export, filter, enumerate, and so on
- Use formatting if you are finished manipulating the objects and are ready to display output on the screen

**Question:** What custom column keys are allowed in **Format-Table** that are not allowed in the calculated properties of **Select-Object**?

## Lesson 3

# Redirecting Formatted Output

In this lesson, you will learn how to redirect formatted output to locations other than the screen. You will also learn about the rules for redirection, and how to display data in other forms.

### Lesson Objectives

After completing this lesson, students will be able to:

- Display and describe the output of a format command
- Redirect formatted output to a file or printer
- Display data in a grid view

### The Output of a Format Command

Like all Windows PowerShell commands that produce output, the format commands produce objects that go to the pipeline. However, unlike most commands, the format commands produce a very specialized kind of object. Only a few other Windows PowerShell commands understand these specialized formatting objects. Therefore, only a few other commands can receive the output of a format command.

Make sure you always remember these two facts:

- Format commands produce a specialized kind of output.
- Only a few other commands can receive the output of a format command.

- Only a small number of commands can accept the specialized objects produced by the format commands
- A good rule to remember is *format right*, meaning that format commands must come at the end of the command line
- There are only a couple of exceptions

Here is a common mistake made by many Windows PowerShell beginning users:

```
Get-Process |  
Format-Table -Property Name, ID, VM, PM -AutoSize |  
ConvertTo-HTML |  
Out-File C:\Processes.html
```

The intent is to produce an HTML table. However, this command produces unusable output because **ConvertTo-HTML** cannot receive the output of a format command.

### Demonstration: Examining Formatting Output

In this demonstration, you will see how format commands produce a specialized kind of object that cannot be accepted by most other commands.

#### Demonstration Steps

1. Use **Get-Member** to examine the output of **Get-Process**.
2. Use **Get-Member** to examine the output of **Get-Process | Select-Object -Property Name, ID**.

3. Use **Get-Member** to examine the output of **Get-Process | Format-Table -Property Name, ID**.
4. Run **Get-Process | Format-Table -Property Name, ID | Export-Csv Procs.csv**.
5. Open **Procs.csv** in Notepad.

## Redirecting Formatted Output

If formatted output cannot be piped to a **ConvertTo-** or **Export-** command, how can it be redirected?

There are three commands that can accept the output of a format command and direct that output to a particular location:

- **Out-Host** displays output on the screen. This command runs by default at the end of every pipeline.
- **Out-File** directs output to a file.
- **Out-Printer** directs output to a printer.

• Three commands can follow a format command to redirect output:

- **Out-Host** – to the screen
- **Out-File** – to a text file
- **Out-Printer** – to a printer

• These commands can follow any format command on the pipeline

• The redirected content will look exactly as it would have looked on the screen

These commands can all follow a format command on the pipeline. For example:

```
Get-Process |  
Format-Table -Property Name, ID,@{n='VM';e={$PSItem.VM / 1KB};formatString='N2'} |  
Out-File Procs.txt
```

The content of the file or printed page will look exactly as it would have on the screen. These commands do not convert data to another format like CSV or XML. These commands merely take output that usually would have appeared on the screen, and redirect it to a file or printer.

## Demonstration: Redirecting Formatted Output

In this demonstration, you will see how to redirect formatted output to a file.

### Demonstration Steps

1. Run the following command and let the output appear on the screen:

```
Get-Process |  
Format-Table -Property Name, ID,@{n='VM';e={$PSItem.VM / 1KB};formatString='N2'}
```

2. Run the same command again, but redirect the output to a file that is named C:\Procs.txt.
3. Open Procs.txt in Notepad.

## Using Grid Views

Windows PowerShell offers another display option on computers where the Windows PowerShell ISE host application is installed.

The **Out-GridView** command can accept regular objects as input, and will display them in an interactive, graphical grid. The contents of the grid are not updated. But the grid can be sorted and filtered by using on-screen controls. The command cannot accept formatted output.

- **Out-GridView** displays objects in a sortable, filterable grid
- Does not accept formatted output
- Only available if the Windows PowerShell ISE host is installed on the computer

### Demonstration: Using Grid Views

In this demonstration, you will see how to use the **Out-GridView** command.

#### Demonstration Steps

1. Display all running processes in a grid view.
2. Sort the grid on the process ID.
3. Filter the grid so that only processes having the name **svchost** are displayed.
4. Close the grid.
5. Run **Get-Process | Format-Table | Out-GridView**. Notice the error message.

**Question:** If you wanted to include a specified set of object properties in a CSV file, would you use **Select-Object** or **Format-Table**?

# Lab: Formatting Output

## Scenario

Your organization has specific criteria for the content and appearance of management reports. You have to write several commands that produce formatted output so that the output can be used in management reports.

## Objectives

After completing this lab, students will be able to:

- Run commands to produce formatted output
- Write commands that reproduce specified output

## Lab Setup

Estimated Time: 45 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click 10961B-LON-DC1, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Formatting Command Output

### Scenario

In this exercise, you will create various commands that produce formatted output.

The main tasks for this exercise are as follows:

1. Create a formatted display of computer system information
2. Create a formatted table of process information
3. Display a table of processes organized by base priority level
4. Display a list of local TCP/IP routes

► **Task 1: Create a formatted display of computer system information**

1. Remember that you can pipe the output of a command to **Get-Member** to see a list of properties, or to **Format-List** \* to see a list of properties and their values. On the LON-CL1 virtual machine, using this command:

```
Get-CimInstance -Class Win32_ComputerSystem
```

2. Create a list display that includes the following:

- Computer name
- Description
- Domain
- Manufacturer
- Model
- Number of processors
- Installed physical memory in gigabytes (GB)

► **Task 2: Create a formatted table of process information**

- Display a table of running processes. The table must include the following:
  - Process name
  - ID
  - Virtual memory in megabytes (MB) to two decimal places
  - Physical memory in megabytes (MB) to two decimal places

The table should not have additional space between the columns. Redirect the table to a text file that is named **Procs.txt**.

► **Task 3: Display a table of processes organized by base priority level**

1. Display tables of running processes. The tables must appear identical to the one output by **Get-Process**. Using a single command, create one table for each value of the process' **BasePriority** property.

► **Task 4: Display a list of local TCP/IP routes**

1. Using the command **Get-NetRoute**, display a table of TCP/IP routes. The table must include the following:
  - Route address family
  - Route metric
  - Type of route
  - Destination prefix
2. Destination prefix must be right-aligned. The table must not include additional space between columns.

**Results:** After completing this exercise, you will have created various commands that produce formatted output.

## Exercise 2: Reproducing Specified Output

### Scenario

In this exercise, you will be given an example of command output. You must write a command that exactly reproduces the specified output.

The main tasks for this exercise are as follows:

1. Write a command that displays file names and sizes as specified
2. Write a command that displays event log entries as specified
3. To prepare for the next module

#### ► Task 1: Write a command that displays file names and sizes as specified

1. Write a command that will display a list of all files having an .exe file name extension in the **C:\Windows** directory.
2. Your output must look exactly as follows:

Name	Size(KB)
explorer.exe	2,273.76
HelpPane.exe	950.50
DfsrAdmin.exe	231.00
notepad.exe	212.50
regedit.exe	148.00
splwow64.exe	123.00
bfsvc.exe	55.50
hh.exe	17.00
winhlp32.exe	10.50
write.exe	10.50

#### ► Task 2: Write a command that displays event log entries as specified

1. Event log entries have a **TimeGenerated** and **TimeWritten**. The difference between those two times is typically zero. However, it is possible for there to be a delay between the time that the event is generated and the time that it is written into the log.
2. Display the most recent 20 entries from the Security event log. Calculate the difference between the time each event was generated and the time that it was written. Display the list exactly as shown here, with the largest time difference shown first, and the smallest time difference shown last.
3. You will use four commands to perform this task. Your output should be similar to this:

EventID	TimeDifference
4672	00:00:00
4634	00:00:00
4672	00:00:00
4624	00:00:00
4634	00:00:00
4624	00:00:00
4672	00:00:00
4648	00:00:00
4624	00:00:00
4672	00:00:00
4648	00:00:00
4624	00:00:00
{continues}	

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have written commands to reproduce specified output.

**Question:** If you redirected formatted output to a file, is there a command that would let you attach that file to an email message?

## Module Review and Takeaways



**Best Practice:** Always make sure that your command works correctly before you worry about how to format the output. Formatting can be complex, and doing it last will help you avoid many common mistakes.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Error when you try to create a custom table column.	

### Review Question(s)

Question: Where might you use **Out-GridView**?

### Real-world Issues and Scenarios

Although Windows PowerShell's formatting commands are powerful, they are not intended to be a complete report creation tool. If you are trying to produce complex reports, consider storing data in Microsoft® SQL Server®, and using Microsoft SQL Server Reporting Services (SSRS). Microsoft SQL Server 2012 Express is free, and includes SSRS.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 6

## Querying Management Information by Using WMI and CIM

### Contents:

Module Overview	06-1
<b>Lesson 1:</b> Understanding WMI and CIM	06-2
<b>Lesson 2:</b> Querying Data by Using WMI and CIM	06-6
<b>Lesson 3:</b> Making Changes by Using WMI and CIM	06-13
<b>Lab:</b> Working with WMI and CIM	06-17
Module Review and Takeaways	06-21

## Module Overview

In this module, you will learn about two parallel technologies. Windows® Management Instrumentation (WMI) and Common Information Model (CIM) both provide local and remote access to a repository of management information including access to robust information available from the operating system, computer hardware, and installed software.



**Additional Reading:** You can read more about CIM Cmdlets at <http://go.microsoft.com/fwlink/?LinkId=306149>.

### Objectives

After completing this module, students will be able to:

- Explain the differences between WMI and CIM
- Query management information by using WMI and CIM
- Invoke methods by using WMI and CIM

## Lesson 1

# Understanding WMI and CIM

In this lesson, you will learn about the architecture of both WMI and CIM. You will learn about the differences between the two technologies, and learn to select the appropriate technology for a given scenario.

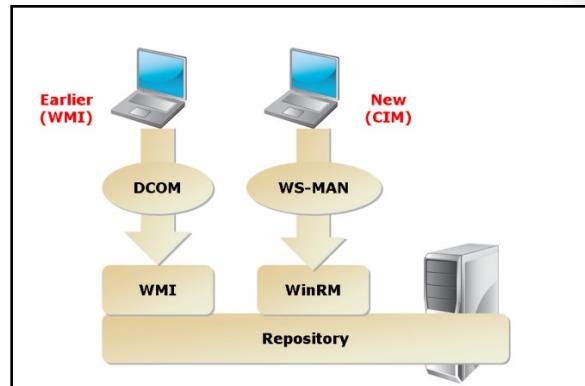
### Lesson Objectives

After completing this lesson, students will be able to:

- Describe the architecture of WMI and CIM
- Explain the purpose of the repository
- Locate online documentation for repository classes

### Architecture and Technologies

Windows Management Instrumentation (WMI) and Common Information Model (CIM) are related technologies. Both are based on industry standards defined by the Distributed Management Task Force (DMTF), which defines independent management standards that can be implemented across different platforms or vendor enterprise environments. WMI is Microsoft's implementation of the Web-Based Enterprise Management (WBEM) standard and is an earlier technology that is based on preliminary standards and proprietary technology. CIM is a newer technology, and is based on open, cross-platform standards. WMI has been available on the Windows operating system since Windows NT 4.0.



Both technologies provide a way to connect to a common information repository (also known as the *WMI repository*). This repository holds management information that you can query and manipulate. Windows PowerShell™ 3.0 and later supports both technologies. Earlier versions of Windows PowerShell support only WMI. In Windows PowerShell 3.0 and later, two parallel sets of commands let you perform tasks by using either WMI or CIM.



**Reference Links:** The Distributed Management Task Force (DMTF) website is available here  
- [www.dmtf.org](http://www.dmtf.org).

### CIM Commands

CIM commands provide many different cross-platform and cross-version capabilities. They support three kinds of connections:

- Connections to the local computer, which always use DCOM.
- Ad hoc connections to a remote computer, which always use the Web Services for Management (WS-MAN) protocol. This protocol is based on HTTP.

- Session-based connections to a remote computer, which can use either DCOM or WS-MAN.

DCOM connections are typically made to the Windows Management Instrumentation (WMI) service that is part of the Windows operating system. WS-MAN connections are made to the Windows Remote Management (WinRM) service, which is the same service that enables Windows PowerShell remoting. You will learn more about remoting in Module 9, “Administering Remote Computers.” WinRM is part of the Windows Management Framework, and is included in Windows Management Framework 2.0 and newer versions. WinRM is installed by default on computers that are running Windows 7, Windows 8, Windows Server® 2008 R2, and Windows Server 2012. Although installed by default on all those operating systems, WinRM and remoting are enabled by default only on Windows 8 and Windows Server 2012.

You can use CIM commands in two ways. The first requires the remote computer to have WinRM installed and enabled. That will typically require that Windows Management Framework 3.0 be installed and that Windows PowerShell remoting be enabled. The second way to use CIM commands is to tell the command to use the earlier WMI technology. That is the same technology used by the WMI commands, and it does not require that Windows Management Framework be installed on the remote computer.

## **WMI Commands**

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

WMI commands do not support session-based connections. The commands support only ad hoc connections over DCOM. Whether used by WMI or CIM commands, DCOM can be difficult to use on some networks. DCOM uses the remote procedure call (RPC) protocol. That protocol requires special firewall exceptions to work correctly.

WMI commands communicate with the WMI service. They do not require any version of the Windows Management Framework on a remote computer, and they do not require that Windows PowerShell remoting be enabled. If the remote computer has the Windows Firewall feature enabled, WMI commands require that the Remote Administration exception be enabled on the remote computer. If the remote computer has a different local firewall enabled, an equivalent exception must be created and enabled.

Because CIM commands can also use DCOM, WMI commands are typically necessary only when you have to make an ad hoc connection to a computer that does not have Windows PowerShell remoting enabled.

## **Should You Use WMI or CIM?**

Most of the time, you should use CIM cmdlets instead of the older WMI cmdlets.

- CIM cmdlets use DCOM when querying the local computer
- CIM cmdlets use WS-MAN for ad hoc connections to remote computers
- CIM cmdlets can use DCOM or WS-MAN for session-based connections to remote computers

The WMI cmdlets can be used when you must make an ad hoc connection to a computer that does not have Windows Management Framework 2.0 or a newer version installed, or to a computer that has Windows Management Framework 2.0 installed but that does not have Windows PowerShell remoting enabled. However, the CIM cmdlets can connect to a computer that does not have Windows Management Framework 2.0 or a newer version installed, and to computers that do not have Windows PowerShell remoting enabled. You must use a **CimSession** to connect to those computers. You will learn about CIM sessions in the next lesson.



**Note:** Microsoft considers the WMI commands within Windows PowerShell to be deprecated, although the underlying WMI repository is still a current technology. Although at the time of this writing, no announcement has been made about the availability of these commands in future product versions, no additional development or improvement is expected for these

commands. You should rely primarily on CIM commands, and use WMI commands only when CIM commands are not practical.

### Commands Instead of Classes

The repository is not well-documented, and discovering the classes that you might need to perform a specific task can be difficult and impractical. The solution is to provide Windows PowerShell commands that behave like any other shell command, but that internally use WMI or CIM. The approach gives you the advantages of Windows PowerShell commands such as discoverability and built-in documentation while also giving you the existing functionality of the repository.

In Windows 8 and Windows Server 2012, Microsoft introduced hundreds of new commands in Windows PowerShell. Many of these commands internally use WMI or CIM. These commands provide better access to the functionality of WMI and CIM so that you can use that functionality without having to deal with their complexity.

## The Repository

The repository used by CIM and WMI is organized into namespaces. A *namespace* is basically a folder, and is used to group 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 may have slightly different namespaces and classes. For example, a domain controller may have a class named ActiveDirectory that would not be present on other computers.

When you work with the repository, you typically work with instances. An *instance* represents an actual occurrence of a class. For example, if your computer has two processor sockets, you will have two instances of the class that represents processors. If your computer does not have an attached tape drive, you will have zero instances of the tape drive class.

Instances are objects, similar to the objects that you have already used in Windows PowerShell. Instances have properties, and some instances have methods. *Properties* describe the attributes of an instance. For example, a network adapter instance might have properties that describe its speed, its 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.

- Namespaces organize related classes
- Classes represent manageable components
- Instances are an actual occurrence of a class
- Instances have
  - Properties that describe the instance's attributes
  - Methods that make the instance perform an action

## Finding Documentation

Class documentation can be difficult to find. Although many Microsoft product groups and independent software vendors (ISVs) expose management information in the repository, few of them create formal documentation. In most cases, an Internet search for a class name will be your best option for finding whatever documentation exists.

The classes in the **root\CIMv2** namespace are an exception. These classes are typically well-documented in the Microsoft Developer Network (MSDN) Library. However, an Internet search is frequently the best way to locate the documentation for a particular class. For example, entering the class name **Win32\_OperatingSystem** in an Internet search engine is the fastest way to locate the documentation webpage for that class.

Remember that both WMI and CIM are not a native part of Windows PowerShell. Instead, they are external technologies that Windows PowerShell can use and understand. However, because they are external technologies, the repository classes are not documented in Windows PowerShell's native Help system.

- Fastest way to find documentation is to enter a repository class name into an Internet search engine
- Classes in the **root\CIMv2** namespace are typically well-documented
- Classes from other namespaces are typically not well-documented

## Demonstration: Finding Classes Documentation

In this demonstration, you will see how to locate online class documentation.

### Demonstration Steps

- Locate the online documentation for the Win32\_BIOS class.

**Question:** Can you think of any situations where you would have to use WMI instead of CIM?

## Lesson 2

# Querying Data by Using WMI and CIM

In this lesson, you will use both WMI and CIM commands to query the repository.

### Lesson Objectives

After completing this lesson, students will be able to:

- Write a command that lists available namespaces
- Retrieve a list of classes from a namespace
- Query instances of a specified class
- Connect to remote computers by using WMI or CIM
- Create and manage CIM sessions

### Listing Namespaces

You can use Windows PowerShell to list all the namespaces on the local computer or on a remote computer. Run this command:

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

 **Note:** It can take several minutes to produce a complete list of namespaces. Notice that **\_\_NAMESPACE** has two underscore characters.

- Listing namespaces helps you discover what the repository on your computer contains

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

- CIM commands offer tab-completion for the **-Namespace** parameter

Windows PowerShell also supports tab completion for namespace names. In this class, you will use only the **root\CIMv2** namespace, which includes all the classes related to the Windows operating system and your computer's hardware. The **root\CIMv2** namespace is the default namespace. Therefore, you do not have to specify the namespace when querying instances from it.

### Demonstration: Listing Namespaces

In this demonstration, you will see how to list local repository namespaces by using WMI.

#### Demonstration Steps

- Use WMI to list all local namespaces.

## Listing Classes

Windows PowerShell can list all the classes in a particular namespace. For example, to list all the classes in the **root\CLIMv2** namespace, run either of these commands:

```
Get-WmiObject -Namespace root\cimv2 -List  
Get-CimClass -Namespace root\CLIMv2
```

 **Note:** It can take several minutes to list all the classes in a large namespace like **root\CLIMv2**.

- Listing classes in alphabetical order can make it easier to decide whether the class you need exists

```
Get-WmiObject -Namespace root\cimv2 -List |  
Sort Name  
  
Get-CimClass -Namespace root\CLIMv2 |  
Sort CimClassName
```

Windows PowerShell supports tab completion of namespace names for CIM commands only. You can type **Get-Cim[tab] –Nam[tab] roo[tab]** to quickly type the second command, pressing the Tab key where [tab] is shown.

Classes are not listed in any particular order. Finding a class can be easier when they are listed alphabetically. For example, if you are looking for a class that represents a process, but do not know the class name, you could quickly move to the "P" section of a sorted list and start to look for the word *process*. To produce an alphabetical list of classes in the **root\CLIMv2** namespace, run either of these commands:

```
Get-WmiObject -Namespace root\cimv2 -List | Sort Name  
Get-CimClass -Namespace root\CLIMv2 | Sort Name
```

 **Note:** In the **root\CLIMv2** namespace you will see classes whose names start with **Win32\_** and other classes whose names start with **CIM\_**. This namespace is the only one where those prefixes are used. Classes starting with **CIM\_** are typically abstract classes. Classes starting with **Win32\_** are typically more specific versions of the abstract classes, and contain information that is specific to the Windows operating system.

Many administrators feel that the repository can be difficult to work with. Finding the class that you must have to perform a given task is basically a guessing game. You have to guess at what the class might be named, and then look through the class list to determine whether you were right. Then you must query the class to determine whether the class contains the information that you need. Because many classes outside the **root\CLIMv2** namespace are not well-documented, this is your best approach. An administrator who is good at using WMI and CIM is also good at making educated guesses.

There is no central directory of repository classes. The repository does not include a search system. You can use Windows PowerShell to perform a basic keyword search of repository class names. For example, to find all classes in the **root\CLIMv2** namespace having *network* in the class name, use this:

```
Get-CimClass -Namespace root\CLIMv2 | Where CimClassName -like '*network*' | Sort  
CimClassName
```

However, this technique cannot search class descriptions, because that information is not stored in the repository. Frequently, an Internet search engine will provide a better way to search for possible class names.



**Note:** You may see some classes whose names begin with two underscore characters (\_). These are *system classes* and they are used internally by WMI and CIM.

**Reference Links:** A graphical WMI Explorer tool, written in Windows PowerShell script, is available from <http://go.microsoft.com/fwlink/?LinkId=306150>. This tool can make it easier to explore the WMI classes that are available on a given computer.

## Demonstration: Listing Classes

In this demonstration, you will see how to list the classes in a namespace.

### Demonstration Steps

1. List the classes in the **root\SecurityCenter2** namespace.
2. List the classes in the **root\CLIMv2** namespace. Sort the list by name.

## Querying Instances

When you know what class you want to query, Windows PowerShell can retrieve class instances for you. For example, if you wanted to retrieve all instances of the **Win32\_LogicalDisk** class from the **root\CLIMv2** namespace, you could run either of the following commands:

```
Get-WmiObject -Class Win32_LogicalDisk  
Get-CimInstance -ClassName  
Win32_LogicalDisk
```

- Query by specifying a class name
- Include **-Namespace** if class is not in **root\CLIMv2**
- Include **-Filter** to restrict the instances that the command returns

```
Get-WmiObject -Class Win32_LogicalDisk  
-Filter "DriveType=3"
```

```
Get-CimInstance -ClassName Win32_LogicalDisk  
-Filter "DriveType=3"
```

- Filter operators are different from Windows PowerShell comparison operators



**Note:** The default configuration settings in Windows PowerShell cause these two commands to have differently formatted output.

Both the **-Class** parameter of **Get-WmiObject**, and the **-ClassName** parameter of **Get-CimInstance**, are positional. That means the following commands work the same way:

```
Get-WmiObject Win32_LogicalDisk  
Get-CimInstance Win32_LogicalDisk
```

By default, both commands retrieve all available instances of the specified class. You can specify filter criteria to retrieve a smaller set of instances. The filter languages used by these commands do not use Windows PowerShell comparison operators. Instead, traditional programming operators are used, as shown in the following table.

Comparison	WMI and CIM operator	Windows PowerShell operator
Equality	=	-eq
Inequality	<>	-ne

Comparison	WMI and CIM operator	Windows PowerShell operator
Greater than	>	-gt
Less than	<	-lt
Wildcard string match	LIKE, with % as wildcard	-like, with * as wildcard
Less than or equal to	<=	-le
Greater than or equal to	>=	-ge
Boolean AND	AND	-and
Boolean OR	OR	-or

For example, to retrieve only the instances of **Win32\_LogicalDisk**, where 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:** WMI and CIM require that string values within the filter criteria be contained within single quotation marks. For example, **Name='BITS'**. For this reason, the whole filter value must be contained within double quotation marks. Using double quotation marks for the entire filter value ensures that the enclosed single quotation marks are correctly sent to WMI or CIM. Date values are also enclosed within single quotation marks.

 **Note:** Many class properties use integers to represent different kinds of things. For example, in the **Win32\_LogicalDisk** class, the **DriveType** property of **3** represents a local fixed disk. A **5** represents an optical disk, such as a DVD drive. You will have to examine the class documentation to learn what each value represents.

## Querying by Using WQL

Both WMI and CIM also accept query statements that are written in the WMI Query Language (WQL). These query statements were common in Microsoft Visual Basic® Script Edition (VBScript) scripts. For example:

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

This command syntax makes it easier to reuse existing query statements that you may have, or that you may find in examples written by other people. A detailed examination of WQL is beyond the scope of this course, and you will not cover it further in this course.

## Demonstration: Querying Instances

In this demonstration, you will see several ways to query class instances from the repository.

### Demonstration Steps

1. Use WMI to display all instances of the **Win32\_Service** class.
2. Use CIM to display all instances of the **Win32\_Process** class.
3. Use CIM to display those instances of the **Win32\_LogicalDisk** class having a drive type of 3.
4. Use CIM and a WQL query to display all instances of the **Win32\_NetworkAdapter** class.

## Remote Computers

At this point in this lesson, you have queried only the local computer's repository. Both the WMI and CIM commands are able to connect to a remote computer. When they connect to a remote computer, they can also specify alternative credentials for the connection.



**Note:** You may not specify alternative credentials for connections to the local computer.

- Use **-ComputerName** to query from a remote computer.
- Use **-Credential** to specify an alternate credential for remote connections.

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

- WMI uses DCOM. CIM uses WinRM for ad-hoc connections.

For the WMI commands, use the –

**ComputerName** parameter to specify a remote computer's name or IP address. You can specify multiple computer names, either in a comma-separated list or by providing a parenthetical command that produces a collection of computer names as string objects. Use the **-Credential** parameter to specify an alternative user name. You will be prompted for the password. For example:

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

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

The CIM commands also support a **-ComputerName** parameter and a **-Credential** parameter. Using these creates an ad hoc connection. If you plan to query multiple classes from the same computer, you can achieve better performance by creating a persistent CIM session instead.

Remember that the CIM commands use the WS-MAN protocol for ad hoc connections. This protocol has specific authentication requirements. Between computers in the same domain or in trusting domains, you typically have to provide a computer's name as it appears in Active Directory®. You cannot typically provide an alias name or an IP address. You will learn more about these and other restrictions in Module 9, "Managing Remote Computers." You will also learn how to work around these restrictions.

## Using CIMSessions

A *CIM session* is a persistent connection to a remote computer, made by using the WS-MAN or DCOM protocol. After a session is created, you can use it to process multiple queries for that computer. You will achieve better performance across multiple queries by using a session rather than by using multiple ad hoc connections.

The basic syntax to create a session and store it in a variable is as follows:

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

- Reusable, persistent, authenticated connections to a remote computer
- Create and store in a variable
- Pass to **-CimSession** parameter instead of **-ComputerName** to target the computer in the specified session
- Sessions time out after a period of inactivity
- You can manually close sessions that are no longer needed

You can create multiple sessions at the same time:

```
$s2 = New-CimSession -ComputerName LON-CL1,LON-DC1
```

When you have one or more sessions in a variable, you can send CIM queries to those sessions:

```
Get-CimInstance -CimSession $s2 -ClassName Win32_OperatingSystem
```

Remember that sessions are designed to work best in a domain environment, between computers in the same domain or in trusting domains. If you have to create a session to a non-domain computer, or to a computer in a non-trusted domain, additional configuration is required. You will learn more about those requirements in Module 9, "Managing Remote Computers."

A session option enables you to specify many session options. One option lets you create the session by using DCOM instead of WS-MAN:

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

The first line in the preceding code defines the CIM DCOM session option. The second line defines the session variable using that CIM DCOM session option, and the final line returns data from the remote machine using that defined session.

CIM sessions remain open while being used. A system-wide idle time-out will close an unused session after the specified time. You can also manually close the sessions for a specified remote computer:

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

To close a session that you have stored in a variable, use this:

```
$sess | Remove-CimSession
```

Or close all open sessions:

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



**Note:** The Help for some commands, such as **Get-SmbShare**, state that they support a **-CimSession** parameter. Those commands use CIM internally. When using those commands to query a remote computer, you can provide a CIM session object to the **-CimSession** parameter to connect by means of the existing session.

## Demonstration: Using CIMSessions

In this demonstration, you will see how to query repository classes from remote computers by using CIMSession objects.

### Demonstration Steps

1. Create a CIM session to LON-DC1.
2. Store the sessions in a variable.
3. Using the variable, query the **Win32\_OperatingSystem** class.
4. Close the session.

**Question:** What are the advantages of creating and using CIM sessions instead of ad hoc connections?

## Lesson 3

# Making Changes by Using WMI and CIM

In this lesson, you will learn to use WMI and CIM to make changes by executing methods.

### Lesson Objectives

After completing this lesson, students will be able to:

- Discover methods of repository objects
- Find online method documentation
- Invoke methods of repository objects

### Discovering Methods

You have learned that objects contain members, and that those members consist of properties, methods, and events. To this point in this course, you have used properties of objects. Now you will be able to use methods.

Methods tell an object to perform some action or task. Repository objects that you query with WMI or CIM frequently 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 the settings for a service that include its logon password, name, and start mode.

- Many repository classes include methods
- Methods tell an object to perform a task or action
- Repository class methods typically reconfigure the manageable component that the class represents
- Use **Get-Member** to discover the methods of a class
- The output will not explain how to use a method; you will need documentation for that

If you know the class that represents the manageable component that you want to reconfigure, you can discover the methods of that class by using **Get-Member**:

```
Get-WmiObject -Class Win32_Service | Get-Member
```

The resulting list will display all available methods. Remember that not every class offers methods, so sometimes the list may not include any methods. The output of **Get-Member** does not explain how to use the methods; unless you already know how to use them, you have to find the documentation for the class.

The same technique will not work with **Get-CimInstance**, because the objects returned by that command do not have a complete set of methods. Instead, you would run this command:

```
Get-CimClass -Class Win32_Service | Select-Object -ExpandProperty CimClassMethods
```

There is no efficient way to search across all classes for a given method. To do this in Windows PowerShell, you would have to query every class, pipe each one to **Get-Member**, and search that output for a method name or keyword. This approach would be very time-consuming and impractical. An Internet search engine would provide a faster and easier way to search for classes and methods.

## Finding Method Documentation

Method documentation, if any exists, will be included on the documentation webpage for the repository class. Remember that repository classes are not typically well-documented, especially classes not in the **root\CMv2** namespace.

An Internet search engine is the fastest way to find the documentation for a class. Enter the class name into a search engine, and one of the first few search results will typically be the class documentation page. The documentation page will include a section named **Methods**, where the class methods are listed. You can click any method name to display instructions for using that method.

Methods	
The Win32_Service class has these methods.	
<b>Method</b>	<b>Description</b>
<a href="#">Change</a>	Modifies a service.
<a href="#">ChangeStartMode</a>	Modifies the start mode of a service.
<a href="#">Create</a>	Creates a new service.
<a href="#">Delete</a>	Deletes an existing service.
<a href="#">GetSecurityDescriptor</a>	Returns the security descriptor that controls access to the service. This method is available starting with Windows Vista. <small>Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0: This method is not available.</small>
<a href="#">InterrogateService</a>	Requests that a service update its state to the service manager.
<a href="#">PauseService</a>	Attempts to place a service in the paused state.
<a href="#">ResumeService</a>	Attempts to place a service in the resumed state.
<a href="#">SetSecurityDescriptor</a>	Writes an updated version of the security descriptor that controls access to the service. This method is available starting with Windows Vista. <small>Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0: This method is not available.</small>
<a href="#">StartService</a>	Attempts to place a service into the startup state.
<a href="#">StopService</a>	Places a service in the stopped state.
<a href="#">UserControlService</a>	Attempts to send a user-defined control code to a service.

The main use of the documentation is to determine what arguments each method requires. For example, the **Win32Shutdown()** method of the **Win32\_OperatingSystem** class accepts a single argument. The argument is an integer, and it tells the method what kind of shut down, restart, or other action to take.

## Demonstration: Finding Methods and Documentation

In this demonstration, you will see how to discover class methods and how to find their documentation.

### Demonstration Steps

1. Display the members of the **Win32\_Service** class.
2. Use Windows PowerShell to display the definition for the **Change()** method of **Win32\_Service**.
3. Find online documentation for the **Change()** method of **Win32\_Service**.

## Invoking Methods

When you know the method that you want to use, and you know how to use it, you can invoke the method. There are three ways to invoke a method.

### Invoke-WmiMethod

The **Invoke-WmiMethod** command can be used by itself, or it can accept a WMI object from **Get-WmiObject** by using the pipeline. Here are two examples that will work the same:

```
Get-WmiObject -Class
Win32_OperatingSystem |
Invoke-WmiMethod -Name Win32Shutdown -
Argument 0
Invoke-WmiMethod -Class Win32_OperatingSystem -Name Win32Shutdown -Argument 0
```

- Three ways to invoke a method:
  - **Invoke-WmiMethod**
  - **Invoke-CimMethod**
  - **ForEach-Object**
- Both **Invoke** techniques can be used by themselves or can accept a pipeline object from the corresponding **Get** command
- Return object includes a **ReturnValue** parameter
  - 0 typically means success
  - See documentation for other values

Both **Get-WmiObject** and **Invoke-Method** have a **-ComputerName** parameter that lets you run the method on a remote computer.

MCT USE ONLY. STUDENT USE PROHIBITED

**Invoke-WmiMethod** will not work correctly if the argument list has to contain a NULL value. For example, the **Change()** method of the **Win32\_Service** class accepts several different arguments. You can provide NULL as a value for any argument that you do not want to change. For example, if you want to change only the service's logon password, you provide NULL for the first seven arguments, and a new password for the eighth argument. **Invoke-WmiMethod** does not support the use of those NULL values.

**Invoke-WmiMethod** is a WMI command. That means that it communicates by using the DCOM protocol.

 **Note:** WMI implements a system of *privileges* in addition to usual permissions. A privilege protects especially sensitive operations, such as restarting a computer. In order to execute those sensitive operations, your WMI connection must be enabled for privileges. The **-EnableAllPrivileges** switch of **Get-WmiObject** enables privileges, and must be used when you plan to invoke a sensitive method.

## Invoke-CimMethod

The **Invoke-CimMethod** command resembles **Invoke-WmiMethod**. However, because it is a CIM command, it communicates by using different protocols:

- When you connect to the local repository, it uses DCOM.
- When you connect to a remote computer name, it uses WS-MAN.
- When you use an established **CIMSession**, it uses either DCOM or WS-MAN, depending on how the session was created.

The argument list for **Invoke-CimMethod** is a dictionary object. These objects consist of one or more key-value pairs. The key for each pair is the argument name, and the value for each pair is the corresponding argument value. For example:

```
Invoke-CimMethod -ComputerName LON-DC1 -Class Win32_Process -MethodName Create
-Arguments @{'Path'='Notepad.exe'}
```

To include multiple arguments in the dictionary, use a semicolon (;) to separate each key/value pair. The command can also accept a repository object from **Get-CimInstance**:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name='notepad.exe'" |
Invoke-CimMethod -MethodName Terminate
```

Notice that you do not have to specify the **-Arguments** parameter for methods that do not require any arguments.

If you use **-ComputerName** or **-CIMSession** with **Get-CimInstance**, and pipe the resulting object to **Invoke-CimMethod**, **Invoke-CimMethod** will invoke the method on whatever computer or session the object came from. For example, to terminate a process on a remote computer:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name='notepad.exe'" -Computername
LON-DC1 | Invoke-CimMethod -MethodName Terminate
```

## ForEach-Object

If **Invoke-WmiMethod** and **Invoke-CimMethod** cannot be used, you can retrieve repository objects and enumerate them to execute methods. For example:

```
Get-WmiObject -Class Win32_Service -Filter "Name='MyService'" |
ForEach-Object { $PSItem.Change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }
```

This example changes the logon password of the service named **MyService**. When you use this technique to invoke a method, you must follow these rules:

- The method name must be followed by an open parenthesis. You may not include a space between the method name and the parentheses.
- Parentheses are required even if the method does not accept any arguments.
- Arguments are provided in a comma-separated list.
- Windows PowerShell uses the built-in variable **\$null** to represent the value NULL.

 **Note:** Remember that **\$\_** must be used instead of **\$PSItem** in Windows PowerShell 2.0 and earlier. Windows PowerShell 3.0 and later can use **\$\_** instead of **\$PSItem**, if you prefer.

## Demonstration: Invoking Methods

In this demonstration, you will see how to invoke methods of repository objects.

### Demonstration Steps

1. Using CIM and the **Reboot()** method of **Win32\_OperatingSystem**, restart LON-DC1.
2. Start Windows Paint.
3. Using WMI and the **Terminate()** method of **Win32\_Process**, close Windows Paint.

**Question:** What are some disadvantages of using **ForEach-Object** instead of one of the Invoke commands to invoke a method?

# Lab: Working with 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, students will be able to:

- Query information by using WMI commands
- Query information by using CIM commands
- Invoke methods by using WMI and CIM commands

## Lab Setup

Estimated Time: 45 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - a. User name: **Administrator**
  - b. Password: **Pa\$\$w0rd**
  - c. Domain: **ADATUM**
5. Repeat step 2 for 10961B-LON-CL1.
6. Perform the lab steps on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Querying Information by Using WMI

### Scenario

In this exercise, you will discover repository classes and then use WMI commands to query them.

The main tasks for this exercise are as follows:

1. Query IP addresses
2. Query operating system version information
3. Query computer system hardware information

4. Query service information

► **Task 1: Query IP addresses**

1. An IP address is part of a network adapter's configuration. Using a keyword like **configuration**, find a repository class that lists IP addresses that are being used by the local computer.
2. Using a WMI command and the class that you discovered in the previous step, display a list of all IP addresses received by using DHCP.

► **Task 2: Query operating system version information**

1. Using a keyword like **operating**, find a repository class that lists operating system version information.
2. Display a list of properties for the class that you discovered in the previous step.
3. Write the properties that contain the operating system version, service pack major version, and operating system build number.
4. Using a WMI command and the class that you discovered in step 1, display the local operating system version, build number, and service pack major version.

► **Task 3: Query computer system hardware information**

1. Using a keyword like **system**, find a repository class that contains computer system information.
2. Display a list of properties and property values for the class that you discovered in the previous step.
3. Using the list of properties and a WMI command, display the local computer's manufacturer, model, and total physical memory. Label the column for total physical memory as **RAM**.

► **Task 4: Query service information**

1. Using a keyword like **service**, find a repository class that contains service information.
2. Display a list of properties and property values for the class that you discovered in the previous step.
3. Using the list of properties and a WMI command, display the service name, status (running or stopped), and logon name for all services whose names start with *S*.

**Results:** After completing this lab, you will have queried repository classes by using WMI commands.

## Exercise 2: Querying Information by Using CIM

### Scenario

In this exercise, you will discover new repository classes and query them by using CIM commands.

The main tasks for this exercise are as follows:

1. Query user accounts
2. Query BIOS information
3. Query network adapter configuration information
4. Query user group information

► **Task 1: Query user accounts**

1. Using a WMI command and a keyword like **user**, find a repository class that lists user accounts.

2. Using a CIM command, display a list of properties for the class that you discovered in the previous step.
3. Using a CIM command and the property list, display a list of user accounts in a table. Include columns for the account caption, domain, SID, full name, and name. The full name column may be blank for some or all accounts.

► **Task 2: Query BIOS information**

1. Using a keyword like **bios** and a WMI command, find a repository class that contains BIOS information.
2. Using a CIM command and the class that you discovered in the previous step, display a list of all available BIOS information.

► **Task 3: Query network adapter configuration information**

1. Use a CIM command to display all instances of the local **Win32\_NetworkAdapterConfiguration** class.
2. Use a CIM command to display all instances of the **Win32\_NetworkAdapterConfiguration** class on LON-DC1.

► **Task 4: Query user group information**

1. Using a WMI command and a keyword like **group**, find a class that lists user groups.
2. Using a CIM command, display a list of user groups on LON-DC1.

**Results:** After completing this exercise, you will have queried repository classes by using CIM commands.

## Exercise 3: Invoking Methods

### Scenario

In this exercise, you will use WMI and CIM commands to invoke methods of repository objects.

The main tasks for this exercise are as follows:

1. Invoke a CIM method
2. Invoke a WMI method
3. To prepare for the next module

► **Task 1: Invoke a CIM method**

- Using a CIM command and the **Reboot()** method of **Win32\_OperatingSystem**, restart LON-DC1.

► **Task 2: Invoke a WMI method**

- Switch back to the LON-CL1 virtual machine
- Using WMI commands and the **ChangeStartMode()** of **Win32\_Service**, change the start mode of the **WinRM** service to **Automatic**.

► **Task 3: To prepare for the next module**

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

1. On the host computer, start **Hyper-V Manager**.

MCT USE ONLY STUDENT USE PROHIBITED

2. In the **Virtual Machines** list, right click **10961B-LON-DC1**, and then click **Revert**.
3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for 10961B-LON-CL1.

**Results:** After completing this exercise, you will have used CIM and WMI commands to invoke methods of repository objects.

**Question:** One of your lab tasks directed you to query **Win32\_Product**. Do you know of any disadvantages when you use this class?

**Question:** What are the main differences between WMI and CIM?

# Module Review and Takeaways



**Best Practice:** Use CIM commands when possible. Unlike WMI commands, the CIM commands offer better performance and are the commands that Microsoft continues to develop and improve over time.

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
"RPC server not found error" when you use WMI commands	
Errors when you use CIM to connect to a remote computer by using the WS-MAN protocol	
"Access Denied" error when attempting to connect to a remote computer.	

## Review Question(s)

**Question:** What do you think is the most difficult part about working with WMI and CIM?

## Real-world Issues and Scenarios

Not all organizations have deployed Windows Management Framework 2.0 or newer versions and enabled Windows PowerShell remoting. That means the CIM commands cannot be used for ad hoc connections in those environments. Although you could create CIM sessions that use the DCOM protocol in those environments, doing this requires additional steps. Many environments will continue to use the WMI commands until remoting is enabled throughout their environment.

## Tools

Another way to explore the repository is to use a graphical tool. One tool is the **PowerShell Scriptomatic**, available at <http://go.microsoft.com/fwlink/?LinkID=306152>.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 7

## Preparing for Scripting

### Contents:

Module Overview	07-1
Lesson 1: Using Variables	07-2
Lesson 2: Scripting Security	07-9
Lab: Working with Security in Windows PowerShell	07-14
Module Review and Takeaways	07-16

## Module Overview

In this module, you will prepare yourself and Windows PowerShell™ to write and run scripts. Although you may have used variables in previous modules of this course, you will now learn about their rules and correct use. You will also learn about the shell's built-in security measures for scripts.



**Additional Reading:** You can read more about Scripting with Windows PowerShell, and find an online community script repository, at <http://go.microsoft.com/fwlink/?LinkID=306153>.

### Objectives

After completing this module, students will be able to:

- Create, use, and manage variables
- Configure shell scripting security

## Lesson 1

# Using Variables

Previous modules in this course introduced variables as a temporary storage container for objects. In this lesson, you will learn more about variables and all the rules for using them.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of variables
- Describe the rules for using variables
- Create, manage, and use variables
- Write mathematical calculations and display information by using variables

### What Are Variables?

In Windows PowerShell, *variables* are temporary storage containers for objects. Each variable can contain one or more objects. A variable that contains more than one object is typically called an *array*.

 **Note:** In some programming languages, array variables and collections of objects have important differences. Those differences are very minor in Windows PowerShell, and the terms *array* and *collection* are sometimes used interchangeably.

- Named, temporary storage containers for objects
- Can contain more than one object, making the variable an *array*
- Stored in the drive **VARIABLE:**
- Commands exist to manage variables
  - **Get-Command –Noun Variable**
  - Ad hoc management is typical
- Variables are scoped and exist only for the duration of their containing scope

The shell stores variables in the PSDrive **VARIABLE:**. You can use that drive to change variables, view them, and remove them. The shell also provides a set of commands that manage variables. To see a list, run **Get-Command –Noun Variable**. Most Windows PowerShell users do not use those commands, because the shell also supports ad hoc variable creation and management. You will see the ad hoc technique in this lesson.

Every time that you start a new Windows PowerShell session, it has its own drive **VARIABLE:**. That drive is populated with several built-in variables that control different parts of the shell's behavior. Some of those built-in variables are *constants*, meaning they are read-only and cannot be changed. For example, the built-in **\$ShellId** variable is read-only. The built-in **\$VerbosePreference** is not read-only, and can be changed.

Variables are scoped. That means that they can be created inside a specific in-memory *container*, and they exist only as long as their container exists. Variables created at the command prompt are stored in the global scope, and exist only until the current shell session is closed. Variables created in a script exist only until that script finishes running. Variables created inside a function exist only until that function finishes running.

## Variable Rules

Variables must have a name. Typically, names consist of letters, numbers, and underscores. However, you can also create variable names that contain other characters if you use a special syntax. The following are all examples of enabled variable names:

- **X**
- **ComputerName**
- **Count5**
- **Modified\_Amount**
- **{Server Path}**

- Variable names:
  - Typically contain letters and numbers, and sometimes underscores.
  - Can contain other characters if enclosed in curly braces (not recommended).
- The \$ is not part of a variable name. It instructs the shell to access the contents of the variable. Do not use \$ when you want to manipulate the variable itself.

The last example, **{Server Path}**, is a variable name that contains a character (that is, a space) that would not typically be enabled in a variable name. By enclosing the variable name in curly braces, you can use any character except curly braces. Variable names such as **{Server Path}** are more difficult to type and to read, and are not recommended.

 **Best Practice:** As a best practice, use variable names that contain only letters and numbers. Use underscores only when necessary. Avoid creating variable names that use curly braces {} to enclose a name that contains spaces or other characters.

You will typically see variable names preceded by a dollar sign (\$). The dollar sign is not part of the variable name. The dollar sign tells Windows PowerShell that you want to access the contents of the variable, instead of working with the variable itself. For example:

```
$var = 'daily'  
Set-Variable -Name $var -Value 700
```

In this example, the shell will set the variable **daily** to the value **700**. The value of **\$var** remains "daily" after the second command is run.

This behavior typically confuses beginning users, who expect the shell to set **var** to **700**. The dollar sign tells the shell to access the contents of **var**, which is **daily**. The **-Name** parameter therefore receives **daily**, and so that is the variable it sets.

## Using Variables

Using variables is straightforward. To assign objects to a variable, use the `=` assignment operator. Remember that the equal sign in Windows PowerShell is not a comparison operator for equality. That operator is `-eq`. The equal sign is used only for assignment. Whatever appears on the right side of the equal sign is executed, and its result will be stored in whatever is located on the left side of the equal sign:

```
$services = Get-Service | Where Status -eq 'Running' | Sort Name -Descending  
$today = Get-Date
```

- Use `=` to assign objects to a variable
- Use index numbers like `[0]` to access individual objects in an array
- Use the period `(.)` to access members of an object instead of the entire object
- Variables can be given a type
  - Use type accelerators like `[string]` and `[int]`
  - Use `-is` and `-as` operators
- Be careful of common mistakes

 **Note:** You can run **Get-Variable**, or run **Dir VARIABLE:**, to list all variables. The **Clear-Variable** command will remove a variable's value, and you can use the **Remove-Item** command in the **VARIABLE:** drive to delete a variable.

## Using Arrays

A variable that contains more than one object is an array. In the previous example, `$services` is an array. To access individual objects in an array, use an index number in square brackets. For example:

```
PS C:\> $services = Get-Service | Where Status -eq 'Running' | Sort Name -Descending  
PS C:\> # First object in the array  
PS C:\> $services[0]  
Status    Name          DisplayName  
----      --          -----  
Running   WSearch      Windows Search  
PS C:\> # Second object  
PS C:\> $services[1]  
Status    Name          DisplayName  
----      --          -----  
Running   wscsvc       Security Center  
PS C:\> # Last object  
PS C:\> $services[-1]  
Status    Name          DisplayName  
----      --          -----  
Running   Appinfo      Application Information  
PS C:\> # Just the name of the third object  
PS C:\> $services[2].Name  
WinRM  
PS C:\> # Just the status of the second-to-last object  
PS C:\> $services[-2].Status  
Running
```

## Accessing Members

Almost everything in Windows PowerShell is an object, and objects have members. *Members* can include methods and properties. When a variable contains an object, accessing just the variable returns the whole object:

```
PS C:\> $today = Get-Date  
PS C:\> $today  
Thursday, January 10, 2013 8:33:16 AM
```

You can use the member resolution operator, which is a period, to access individual members of an object:

```
PS C:\> $today.DayOfWeek
Thursday
PS C:\> $today.AddDays(90)
Wednesday, April 10, 2013 8:33:16 AM
```

## Giving Variables a Type

Windows PowerShell does not require you to specify an object type for a variable. For example:

```
PS C:\> $x = Get-Service
PS C:\> $x = Get-Process
```

Those two commands will run without error. After the first command, **X** contains a collection of **System.ServiceProcess.ServiceController** objects. After the second command, **X** changes to contain a collection of **System.Diagnostics.Process** objects.

You can specify a type for variables. When you do this, the shell will not let objects of another type be stored in the variable. Common types include the following:

- **[string]** for string objects
- **[Boolean]** for Boolean values
- **[int]** for integer values

For example:

```
PS C:\> [int]$x = 100
PS C:\> $x = 'Hello'
Cannot convert value "Hello" to type "System.Int32". Error: "Input string was not in
a correct format."
```

This example shows an error, because the variable **X** can no longer contain a **String** object. It can contain only integers, and the string **Hello** could not be converted to an integer. You can explicitly retype a variable:

```
PS C:\> [int]$x = 100
PS C:\> [string]$x = 'Hello'
PS C:\> $x
Hello
```

You can also use the **-is** operator to determine whether a variable contains objects of a specified type, and the **-as** operator to convert an object to a different type. For example:

```
PS C:\> [int]$x = 100
PS C:\> [string]$x = 'Hello'
PS C:\> $x
Hello
PS C:\> $z = 'Hello'
PS C:\> $y = '5'
PS C:\> $z -is [string]
True
PS C:\> $y -is [string]
True
PS C:\> $y2 = $y -as [int]
PS C:\> $y2 | Get-Member
TypeName: System.Int32
```

That example shows both the **-is** and **-as** operators. You will learn more about these in Module 12, “Using Profiles and Advanced Windows PowerShell Techniques.”

## Common Mistakes

Windows PowerShell beginning users will frequently forget that an object is a complex data structure that consist of many members. For example, the following is a common mistake:

```
$file = Get-ChildItem -Path C:\Examples -File  
Get-Content -Path $file
```

This example includes two mistakes:

- **\$file** does not necessarily contain only one object. It may contain multiple objects, that is, if there are multiple files present. **Get-Content** can retrieve the content from multiple files at the same time, but that may not be the intended result.
- **\$file** contains all of a **FileInfo** object, but the **-Path** parameter of **Get-Content** accepts only a **String** object.

The example might work correctly if rewritten as follows:

```
$file = Get-ChildItem -Path C:\Examples -File  
Get-Content -Path $file.FullName
```

This revised example would work correctly in Windows PowerShell 3.0 or later. The variable **file** actually contains a collection of objects, and the collection does not have a **FullName** property. In earlier versions of Windows PowerShell, this example would fail. In Windows PowerShell 3.0 and later, the example works because the shell assumes that you intended to access the **FullName** property of each object in the collection, instead of the collection itself.

## Math Operators and Quotation Marks

Windows PowerShell includes a complete set of mathematical operators. The four basic operators are as follows:

- **+** for addition and string concatenation
- **-** for subtraction
- **\*** for multiplication
- **/** for division

You can also use parentheses to control the order of evaluation in a mathematical expression. These operators can be used with variables:

```
PS C:\> $x = 5  
PS C:\> $y = 5  
PS C:\> $x + $y  
10  
PS C:\> $x * $y  
25
```

- Math operators:
  - **+** **-** **\*** **/**
  - **+** is also used for string concatenation
- In single quotation marks, variables are left as literal values
- In double quotation marks, variables are replaced with their contents
- Subexpressions like **\$(**\$service[0].name**)** are also replaced inside double quotation marks

## Variables in Strings

When a string is enclosed in single quotation marks, Windows PowerShell does not evaluate the contents of the string. For example:

```
PS C:\> $example = 'Windows PowerShell'  
PS C:\> $string = 'The example is $example'
```

```
PS C:\> $string
The example is $example
```

However, when the string is enclosed in double quotation marks, the shell looks for the dollar sign character. When it finds one, it assumes that the following characters are a variable name, and it replaces the variable with its contents. For example:

```
PS C:\> $example = 'Windows PowerShell'
PS C:\> $string = "The example is $example"
PS C:\> $string
The example is Windows PowerShell
```

 **Note:** Windows PowerShell also evaluates the grave accent (`) in double quotation marks. When the grave accent is placed before any other character, the grave accent will remove that character's special meaning, if it has one. For example, `\$x will be treated as the literal value \$x because the grave accent removes the special meaning of \$. For some characters, including t and n, the grave accent assigns special meaning. For example, `t represents a tab character, and `n represents a new line character.

For more details about grave accents, see [help about\\_escape\\_characters](#).

## Subexpressions in Strings

Variable replacement works only for the variable itself. You cannot access a member of a variable by using that same syntax. For example:

```
PS C:\> $process = Get-Process
PS C:\> $string = "The process name is $process[0].name"
PS C:\> $string
The process name is System.Diagnostics.Process (conhost) System.Diagnostics.Process
(csrss) System.Diagnostics.Process
(csrss) System.Diagnostics.Process (devenv) System.Diagnostics.Process (dwm)
System.Diagnostics.Process (explorer) Syst
em.Diagnostics.Process (Idle) System.Diagnostics.Process (iexplore)
System.Diagnostics.Process (iexplore) System.Diagno
stics.Process (lsass) System.Diagnostics.Process (msdtc) System.Diagnostics.Process
(MsMpEng) System.Diagnostics.Proces
s (OSPPSVC) System.Diagnostics.Process (powershell) System.Diagnostics.Process
(powershell_ise) System.Diagnostics.Pro
cess (SearchIndexer) System.Diagnostics.Process (services) System.Diagnostics.Process
(smss) System.Diagnostics.Process
(splwow64) System.Diagnostics.Process (spoolsv) System.Diagnostics.Process (svchost)
System.Diagnostics.Process (svchost)[0].name
```

In this example, **\$process** was replaced. When Windows PowerShell 3.0 or later is asked to display a collection of objects, it displays their name so that the results show the name of every process. Notice that the end of the results includes **[0].name**. That portion of the code was not used as part of the replacement. This example shows a common mistake made by beginning users. The correct way to perform this task is to use a subexpression:

```
PS C:\> $process = Get-Process
PS C:\> $string = "The process name is $($process[0].name)"
PS C:\> $string
The process name is conhost
```

The subexpression consists of **\$()**. Everything inside the parentheses is treated as executable code, and the subexpression is replaced with the result of that executable code.



**Best Practice:** By using variables, subexpressions, and double quotation marks, you should never have to use + as a string concatenation operator. For example, instead of `$result = "The first service is named " + $services[0].name`, you should use `$result = "The first service is named $($services[0].name)"`. Subexpressions enclosed in double quotation marks are typically easier to read than concatenated string expressions.

## Demonstration: Using Variables

In this demonstration, you will see several ways to use variables.

### Demonstration Steps

1. Assign **100** to the variable **x1**.
2. Put a list of all running processes in the variable **procs**.
3. Use the variable **procs** to display the name of the first running process.
4. Use the variable **procs** to display a list of processes, sorted in descending order of virtual memory use.
5. Using double quotation marks and the variable **x1**, display the phrase **The content of x1 is 100**.
6. Assign **100** to the variable **x2**.
7. Using the variables **x1** and **x2**, display the result of 100 multiplied by 100.
8. Using the variable **procs** and double quotation marks, display the name and CPU usage of the first running process.
9. Using the variable **x2**, double quotation marks, and the grave accent, display the phrase **\$x2 contains 100**.
10. Using the variable **procs**, display the name of all running processes.
11. Using the variable **procs**, display all properties and property values for the first running process.
12. Display a list of members for the first object in **procs**.

**Question:** What happens to variables when you close the shell?

## Lesson 2

# Scripting Security

In this lesson, you will learn about Windows PowerShell's features for scripting security. You will also learn about the shell's security goals.

### Lesson Objectives

After completing this lesson, students will be able to:

- Describe Windows PowerShell's security goals
- Configure the shell execution policy
- Explain the role of trust in scripting security
- Describe additional shell security features

### Security Goals

Before you start to explore Windows PowerShell's security features, you should understand the shell's objectives and goals for security.

Security in Windows PowerShell is intended to *help* slow down an uninformed user if that user is unintentionally trying to run an untrusted script. This statement about the extent of security in Windows PowerShell is very clear. The shell is *not* intended to:

- Absolutely stop anyone from doing anything
- Slow down or stop an informed user from taking a deliberate action
- Act as an antimalware feature

- Shell security features cover a very limited set of goals:
  - To slow down...
  - An uninformed user...
  - Who is unintentionally...
  - Trying to run an untrusted script

- The shell is not antimalware replacement
- The shell is not intended to...
  - Stop anyone
  - Slow down an informed user
  - Inhibit a deliberate action

Thus, the scope of the shell's security features is very specific. The shell does not eliminate the need for antimalware software. The shell does not reduce the need for good security practices and configurations. The shell does not eliminate the need to educate users about good security practices. The shell's security features are intended to be part of a multilayer security strategy, following the principle of *defense in depth*.

One example of what the shell's security features protect against are the many different viruses that were based on Microsoft® Visual Basic® Scripting Edition (VBScript) in the early 1990s. Those viruses were typically spread through emails. For example, a script named "Postcard from your Mother.vbs" would be included as an email attachment. Unsuspecting users would open the script expecting a message from a parent, and instead the script would execute. Windows PowerShell's features are well designed to help prevent that specific kind of attack.

MCT USE ONLY. STUDENT USE PROHIBITED

## Execution Policy

The shell's primary security feature is its execution policy. By default, this policy is set to **Restricted**. You can run **Get-ExecutionPolicy** to see the current execution policy setting.

The policy can be controlled in one of three ways:

- Run **Set-ExecutionPolicy** as a local administrator to set the execution policy for the computer. This is a *persistent setting*, meaning that it will remain in effect for future shell sessions.
- Use a Group Policy Object (GPO) to set the execution policy for domain computers. A GPO setting will override any local setting and will affect all shell sessions for the specified users or computers.
- Run **PowerShell.exe** by using the **-ExecutionPolicy** parameter. This parameter will override local and GPO settings, and will affect only that shell session.

Five execution policy settings:

- **Restricted** (default)
- **AllSigned**
- **RemoteSigned**
- **Unrestricted**
- **Bypass**

Three ways to change:

- **Set-ExecutionPolicy** command
- Group Policy Object
- **-ExecutionPolicy** parameter of **PowerShell.exe**



**Note:** Notice that any user can run **PowerShell.exe** and use the **-ExecutionPolicy** parameter. That means any informed user can deliberately override both the local execution policy and any GPO setting. This behavior is consistent with the security goals outlined earlier, because those goals do not extend to an informed user taking deliberate action.

The execution policy has five possible settings:

- **Restricted** is the default, and prevents any scripts from running.
- **AllSigned** lets you run only those scripts that carry an intact digital signature, assuming that signature was created by using a certificate issued by a trusted certificate authority (CA).
- **RemoteSigned** lets all scripts stored on the local computer run, and lets other scripts run only if they carry an intact digital signature, assuming that signature was created by using a certificate issued by a trusted CA.
- **Unrestricted** allows all scripts to run.
- **Bypass** is a special setting. It is intended to be used by applications that host Windows PowerShell. This setting disables shell security and enables the application to handle script security.

Digital signatures can be added to any script by using the **Set-AuthenticodeSignature** command. The command requires you to install a Class 3 digital certificate on your computer. These certificates are also known as *code-signing certificates*. Your certificate must be compatible with Microsoft Authenticode technology.



**Note:** Code-signing certificates differ from Class 1 certificates that are used to sign or encrypt email. Class 3 certificates typically require better proof of identity.

You can use certificates issued by a trusted internal CA or by a trusted commercial CA.

## Understanding Trust

The **AllSigned** and **RemoteSigned** execution policy settings rely on certificates issued by a trusted CA. So what does *trust* mean?

Any digital certificate, including a code-signing certificate, is a form of digital identification (ID). In other words, the certificate attests to your identity, just as a driver's license or passport attests to your identity. Any form of identification is only as good as the process that was used to verify your identity before issuing the identification.

Most commercial CAs have a similar process for verifying your identity before issuing a code-signing certificate. Most CAs issue these certificates only to organizations, and do not issue them to individuals. They typically verify the organization's identity by using public documents, corporate credit services, and other independent sources. When a CA uses a thorough, reliable process to verify applicant identities, that CA is said to be *trusted*.

For example, suppose that a CA named Contoso exists. Your organization might review the process that Contoso uses for identity verification. If your organization agreed that the process was thorough and reliable, your organization could decide to trust Contoso. Doing this would let your organization to accept and recognize all certificates issued by Contoso.

A signed script is not guaranteed to be free of malware. However, if the certificate used to sign the script was issued by a trusted CA, you can identify the author of the script by using the script's signature. If the script does contain malware, the author will be known to you, and you can take appropriate actions against the author. Because the CA verified their identity, you know who the author is. The script is no longer anonymous.

A digital signature also guarantees the authenticity of the script. After signed, the script cannot be changed in the slightest way without breaking the signature. A broken signature indicates that the script was changed by someone other than the original author. A script with a broken signature is not trusted.

The critical element to trust is the identity verification process used by the CA that issued the certificate. If a CA uses an unreliable process, that CA could be tricked into issuing false identification to a malicious script author. You would be unable to correctly identify the author of such a script. As a security practice, your organization should periodically review the processes of your trusted CAs, and make sure that each CA's process is acceptable.

- A digital certificate is a form of electronic identification
- The certificate is only reliable if it was issued by using a reliable identity verification process
- A trusted Certification Authority (CA) is one that uses a reliable process to verify identities of certificate applicants
- A trusted script is one that has been digitally signed by using a trusted certificate; the author of a trusted script is therefore known to you

MCT  
ICE ONLY  
STUDENT USE PROHIBITED

## Other Security Features

In addition to the execution policy, Windows PowerShell has other features that are designed to increase security.

The file name extensions used for Windows PowerShell script files are not registered with the operating system as executable file types. By default, they are associated with Notepad. If you double-click a script file, it will therefore open in Notepad, not run in Windows PowerShell. File name extensions include the following:

- .ps1 for script files
- .psm1 for script modules
- .psd1 for module manifests
- .format.ps1xml for view definitions
- .ps1xml for other XML files, including type extensions
- .psc for console configuration files
- .pssc for session configuration files

- Filename extensions are not executable and are not associated with Windows PowerShell
- Scripts must be run by using a relative or absolute path

Windows PowerShell does not search the current folder for scripts. If you are working in the C directory, and it contains a script named Example.ps1, you cannot run the script by using this command:

```
PS C:\> Example
```

You must provide either a relative or absolute path to run the script. Both examples are correct:

```
PS C:\> ./Example  
PS C:\> C:\Example
```

The first example uses a relative path, where `./` specifies the current directory. The second example uses an absolute path. This syntax differs from commands that are loaded in memory, because commands are not preceded by a path.



**Note:** Tab completion can make it easier to run a script that is in the current directory. Type all or part of the script's file name, and press the Tab key. Windows PowerShell will complete the file name and add a relative path. You can then press Enter to run the script.

## Demonstration: Shell Security

In this demonstration, you will see how to set the local execution policy.

### Demonstration Steps

1. Determine the current execution policy.
2. Set the local execution policy to **Restricted**.
3. Run a non-code-signed file E:\Mod07\Democode\Test-NonCodesignedFile.ps1 and verify functionality.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Change execution policy to **RemoteSigned**.
5. Run code-signed file **E:\Mod07\Democode\Test-CodesignedFile.ps1**.
6. Verify file runs successfully.

**Question:** What happens if you try to run a script that exists in the current directory, but you do not provide an absolute or relative path?

# Lab: Working with Security in Windows PowerShell

## Scenario

You have to configure the Windows PowerShell execution policy on your local computer.

## Objectives

After completing this lab, students will be able to:

- Configure the local execution policy on a computer.

## Lab Setup

Estimated Time: 10 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

- On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
- In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
- In the Actions pane, click **Connect**. Wait until the virtual machine starts.
- Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
- Repeat steps 2 through 4 for 10961B-LON-CL1.
- Perform the lab steps on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Configure Security

### Scenario

In this exercise, you will configure and test the execution policy on a computer.

The main tasks for this exercise are as follows:

- Review the execution policy
- Run a script
- To prepare for the next module

#### ► Task 1: Review the execution policy

- Review the current execution policy setting.

► **Task 2: Run a script**

1. Create a script file named **C:\Test.ps1** that contains the command **Get-Service**.
2. Attempt to run the newly created non-code-signed script.
3. Set the execution policy to **RemoteSigned**.
4. Now attempt to run the code-signed file **E:\Mod07\Democode\Test-CodesignedFile.ps1** and verify the file runs successfully.

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have configured and tested the execution policy on a computer.

**Question:** Will the execution policy remain in effect?

**Question:** What is the best way to manage execution policy in an enterprise environment?

## Module Review and Takeaways



**Best Practice:** Use Group Policy to configure the execution policy in your environment. Group Policy offers a centralized and easier way to consistently configure this important security setting.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Script will not run.	

### Review Question(s)

**Question:** Do organizations typically use a single execution policy throughout their environment?

### Real-world Issues and Scenarios

If you try to change the local execution policy, but a Group Policy Object (GPO) is already applying an execution policy setting, the GPO will override your local change. Windows PowerShell will display a warning so that you know your change will not have any effect.

# Module 8

## Moving from Command to Script to Module

### Contents:

Module Overview	08-1
<b>Lesson 1:</b> Moving from Command to Script	08-2
<b>Lab A:</b> Moving from Command to Script	08-7
<b>Lesson 2:</b> Moving from Script to Function to Module	08-11
<b>Lab B:</b> Moving from Script to Function to Module	08-16
<b>Lesson 3:</b> Implementing Basic Error Handling	08-20
<b>Lab C:</b> Implementing Basic Error Handling	08-24
<b>Lesson 4:</b> Using Basic Scripting Constructs	08-26
<b>Lab D:</b> Creating an Advanced Function	08-30
<b>Lesson 5:</b> Exploring Other Scripting Features	08-34
Module Review and Takeaways	08-36

## Module Overview

In this module, you will learn to package a Windows PowerShell™ command in a script module. Script modules are an effective way to share commands with coworkers and colleagues. With script modules, you can make complex commands available to someone with less technical expertise, and you can more easily reuse the commands.



**Additional Reading:** Some of the techniques in this module are further documented at the TechNet about\_Functions\_Advanced page, <http://go.microsoft.com/fwlink/?LinkId=306154>.

This module is intended as an introduction to scripting and toolmaking in Windows PowerShell.

### Objectives

After completing this module, students will be able to:

- Improve a stand-alone command into a script
- Improve a script into a function and script module
- Implement basic error handling
- Implement basic scripting constructs
- Explain additional Windows PowerShell scripting features

## Lesson 1

# Moving from Command to Script

In this lesson, you will be given a Windows PowerShell command that works perfectly. You will learn to identify its changing values, create parameters for those values, and create and test a parameterized script that implements the command.

### Lesson Objectives

After completing this lesson, students will be able to:

- Test a working command
- Identify values in a command that might change every time the command is used
- Create parameters to stand in for changing command values
- Test a parameterized script
- Document a script by adding comment-based help

### Start with a Working Command

Any time that you start to work in a script, you start to work with a greater level of complexity. Scripts include many components, and each of those can potentially introduce an error or bug. To reduce how long you have to spend debugging a script, you should test individual commands in the Windows PowerShell console first. If you have each command working individually by using test values, you can paste that command into a script and probably have fewer problems with it.

- Start with a single command
- Use test values to verify that the command syntax and output is correct

```
Get-EventLog -LogName Security -ComputerName localhost |  
Where EventID -eq 4624 |  
Select -First 50
```

For example, when you start with a command, provide test values for each parameter. Then you can run the command by itself in the console. You might use **localhost** for a computer name, even though your final script might not use **localhost**.

For example, suppose that you are asked to create a tool that can be used by your Help Desk. The tool must retrieve the 50 most recent Security event log entries from a designated computer. The tool must retrieve only event log entries that have a specified event identification (ID). You might create the following command as the basis for this new tool:

```
Get-EventLog -LogName Security -ComputerName localhost |  
Where EventID -eq 4624 |  
Select -First 50
```

This command uses values such as **localhost** and **4624** that will not always be used when the final tool is run. By starting with these test values, you can make sure that the command works correctly.

## Identifying Values that Might Change

When you have the command working correctly, you have to identify the values that might change every time someone else runs the command. Frequently, these will be the same values that you used test values for when you were testing the command.

For example, in the following command are two values that might change:

```
Get-EventLog -LogName Security -ComputerName localhost | Where EventID -eq 4624 | Select -First 50
```

- Identify values in your test command that might need to change each time the command is run

```
Get-EventLog -LogName Security -ComputerName localhost | Where EventID -eq 4624 | Select -First 50
```

 **Note:** The preceding example may require that you manually start the Remote Registry Service if it is not started.

These values are **localhost** and **4624**. Remember that the tool is supposed to retrieve designated event IDs from a specified computer to make both of those values changeable. The tool is always supposed to retrieve event log entries from the Security event log, and is always supposed to retrieve no more than 50 entries. That means the values **Security** and **50** are not changeable. Those can be left alone.

## Parameterizing Changing Values

Now you can start to create a script file.

To take advantage of the newest features in Windows PowerShell, start your script with the **[CmdletBinding()]** attribute. This attribute enables certain Windows PowerShell features in a script, and differentiates the script from earlier versions that do not use those features.

Next, create a **Param()** block. It must come directly after the **[CmdletBinding()]** attribute. Inside the block, you create a comma-separated list of parameters. Each parameter is basically a variable, and so each parameter name starts with a dollar sign (\$). As a best practice, also define the data type for each parameter. Data types include **[string]** for strings, **[int]** for int, **[switch]** for on/off switches, and so on.

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$ComputerName,
    [int]$EventID = 4624
)
Get-EventLog -LogName Security -ComputerName $ComputerName | Where EventID -eq $EventID | Select -First 50
```

 **Note:** You cannot use the **[CmdletBinding()]** attribute unless you also use a **Param()** block. The **Param()** block can be empty if your script will not need to use any input parameters.

You can also define a default value for parameters. A default value is a good idea when the value might have to change only occasionally, but will usually be the same value each time the script is run.

You can also identify some or all parameters as being mandatory. When you identify a parameter as mandatory, Windows PowerShell will prompt for a value when the person running the script does not provide one in advance. Default values are ignored when a parameter is marked as mandatory.

For example, you might now have a script that looks as follows:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$ComputerName,
    [int]$EventID = 4624
)
Get-EventLog -LogName Security -ComputerName $ComputerName |
Where EventID -eq $EventID |
Select -First 50
```

Notice that the parameter names have also been used instead of the old test values. The **Param()** block defines the parameters, and then you can use the parameters instead of the test values in your command.

 **Note:** Your parameter names will appear however you type them in the **Param()** block. Although Windows PowerShell is not case-sensitive, this example used **\$ComputerName** so that the parameter would be displayed and be more consistent with cmdlets that have a – **ComputerName** parameter.

 **Note:** In this example, the script's **\$ComputerName** parameter is being passed as a value to the **-ComputerName** parameter of the **Get-EventLog** command. There is no technical reason for these two to share the same name. However, using **\$ComputerName** makes the script parameter consistent with the parameters of other Windows PowerShell commands.

Windows PowerShell gives you some flexibility in how you format the **Param()** block. The one required rule is that each parameter be separated from the next by a comma. The last parameter is not followed by a comma. The following is acceptable:

```
[CmdletBinding()]
Param([Parameter(Mandatory=$True)] [string]$ComputerName, [int]$EventID = 4624)
```

Most experienced users prefer the first example because it is easier to read and to maintain.

## Demonstration: Parameterizing a Working Command

In this demonstration, you will see how to take an existing command and parameterize changing values in a script.

### Demonstration Steps

1. Start with the file E:\Mod08\Democode\Param1.ps1, which contains an existing, working command.
2. Identify within that script the values that might change.
3. Parameterize changeable values.

## Test the Parameterized Script

After you finish parameterizing the command in your script, you can test the script. In the Windows PowerShell Integrated Scripting Environment (ISE), you should save the script to a file before you run it. Certain script features will not work correctly unless the script is saved to a file.

After you save the script, press F5 or click the **Run script** button in the toolbar to run the script. The script will run in the Console pane. Windows PowerShell prompts you for any mandatory parameters. You can also test the script by manually switching to the Console pane, typing the path and file name of the script, and adding any parameters that you want. Running the script manually is a better way to test it, because you are running the script just as someone else will run it.

- Always save your script before testing.
- After the script is saved, the ISE will automatically save it each time you run it.
- Test your script:
  - By pressing F5 in the ISE. You will be prompted for mandatory parameters, but not for optional ones.
  - By typing the script's path, file name, and parameters in the Console pane and then pressing Enter. This allows you to specify any parameters you want.

## Adding Verbose Output

Many administrators prefer that their scripts display status messages as the script runs. These status messages tell the user when something happens, and can provide diagnostic information when the script does not run correctly. Windows PowerShell provides a command, **Write-Verbose**, that creates these status messages for you.

 **Best Practice:** You may also discover the **Write-Host** command. It is not the correct way to create status messages in a script. **Write-Host** can be used correctly only in a few scenarios. You should avoid using the command if possible.

By default, Windows PowerShell does not display verbose output when you run the script. You must add the **-verbose** parameter when you are running the script to enable the verbose output.

## Demonstration: Adding Verbose Output

In this demonstration, you will see how to add verbose output to a script.

### Demonstration Steps

1. Ensure that the local execution policy is set to **RemoteSigned**.
2. Start with the existing script E:\Mod08\Democode\Param2.ps1.
3. Add verbose output.
4. Run the script and allow for the verbose output to be suppressed.
5. Run the script and display the verbose output.

## Documenting the Script

The **Write-Verbose** command can help document the functionality of a script. When someone else reads the script, the verbose messages can provide information about what each section of the script is supposed to do.

However, most administrators agree that scripts need additional documentation that describes their purpose, identifies their author, and provides examples of how to use them. Windows PowerShell provides a feature known as *comment-based help* that meets this need.

- Provides an in-script documentation block
- Includes:
  - Synopsis
  - Description
  - Parameter information
  - Examples
  - Other documentation
- Run **help about\_comment\_based\_help** to see a full list of documentation features

Comment-based help appears in a comment block at the beginning of the script. It uses predefined keywords to identify sections for a synopsis, a detailed description, parameter documentation, examples, and so on. You can view the feature's documentation, and all available keywords, by running **Help about\_comment\_based\_help – ShowWindow** in Windows PowerShell.

An example of comment-based help looks as follows:

```
<#
 .SYNOPSIS
Retrieves network adapter information from a computer.
.DESCRIPTION
Uses CIM to retrieve information about physical adapters only.
.PARAMETER ComputerName
The name of the computer to query.
.EXAMPLE
.\Get-NetAdapterInfo.ps1 -ComputerName LON-DC1 -Verbose
#>
```



**Note:** The comment-based help keywords like **.SYNOPSIS** and **.EXAMPLE** are not case-sensitive. However, it is a best practice to type them in all uppercase letters.

## Demonstration: Adding Comment-Based Help

In this demonstration, you will see how to add a basic comment-based help block to a script.

### Demonstration Steps

1. Add comment-based help to the script E:\Mod08\Democode\Param3.ps1.
2. Save the script.
3. Display the comment-based help by using the **Help** command.

**Question:** Do **Write-Verbose** and comment-based help serve the purpose of documenting a script and its functionality?

# Lab A: Moving from Command to Script

## Scenario

You have written a Windows PowerShell command and have to share it with several coworkers. Before sharing the command with them, you have to package it in a script. Your coworkers must be able to use this script to run the command by providing only minimal input to the script.

## Objectives

After completing this lab, students will be able to:

- Test a Windows PowerShell command
- Parameterize changing values in a command
- Produce verbose messages from a script
- Document a script by using comment-based help

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - a. User name: **Administrator**
  - b. Password: **Pa\$\$w0rd**
  - c. Domain: **ADATUM**
5. Repeat steps 2 through 4 for **10961B-LON-CL1**.
6. Perform the lab steps on the **10961B-LON-CL1** virtual machine.

## Exercise 1: Test the Command

### Scenario

You are given a working Windows PowerShell command. The command is as follows:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName localhost  
|  
Select-Object -Property @{n='DriveLetter';e={$PSItem.DeviceID}},
```

```
    @{n='FreeSpace(MB)';e="{!!{0:N2}!!" -f ($PSItem.FreeSpace /  
1MB)}},  
    @{n='Size(GB)';e="{!!{0:N2}!!" -f ($PSItem.Size / 1GB)}},  
    @{n='FreePercent';e="{!!{0:N2}!!%" -f ($PSItem.FreeSpace /  
$PSItem.Size * 100)}}}
```

You have to test this command and make sure that it works correctly before you use the command in a script.

If you do not want to type the command, you will find it in E:\Mod08\Labfiles\LabA\Exercise1.ps1.



**Note:** If you try to repeat this exercise on a computer running Windows PowerShell 2.0, remember that `$`\_ must be used instead of `$PSItem`.

The main tasks for this exercise are as follows:

1. Test the Command

► **Task 1: Test the Command**

1. Open Windows PowerShell ISE as Administrator.
2. Test the command and make sure that it runs without error.

**Results:** After completing this exercise, you will have tested the command and verified its functionality.

## Exercise 2: Parameterize Changing Values

### Scenario

In this exercise, you will identify and parameterize the values in a command that might have to change every time someone runs the command. You should be able to identify and parameterize two values in the command that you were given.

The main tasks for this exercise are as follows:

1. Identify changing values in a command
2. Create a new script file to work in
3. Add a parameter block to the script
4. Add parameters and use parameters in the command
5. Test the script

► **Task 1: Identify changing values in a command**

1. In the Windows PowerShell ISE, open **E:\Mod08\Labfiles\LabA\Exercise1.ps1** if it is not already open.
2. Identify two values in this command that might have to change every time someone runs the command.

► **Task 2: Create a new script file to work in**

1. In the Windows PowerShell ISE, create a new script file.
2. Save the empty script file as **C:\Get-DiskInfo.ps1**.

► **Task 3: Add a parameter block to the script**

1. In your blank Get-DiskInfo.ps1 script, create a new parameter block. Make sure that you include the cmdlet binding attribute before the parameter block.
2. Add a mandatory string parameter named **\$ComputerName**.
3. Add an integer parameter named **\$DriveType**.
4. Give the drive type parameter a default value of **3**.
5. Save and close the script.

► **Task 4: Add parameters and use parameters in the command**

1. Copy and paste the command that you were given in the exercise scenario at the beginning of the lab into your Get-DiskInfo.ps1 script. Paste the command below the parameter block. If you are using a printed course book, you may open **E:\Mod08\Labfiles\LabA\exercise1.ps1** and copy and paste from that file.
2. Replace the fixed values **3** and **localhost** with **\$DriveType** and **\$ComputerName** respectively.
3. Save the script.

► **Task 5: Test the script**

1. Run the script. To do so in the ISE, either press F5 or click the **Run script** button in the toolbar.
2. For the **ComputerName** prompt, type **LON-CL1**, and then press Enter.
3. Verify that the script finishes without error.

**Results:** After completing this exercise, you will have identified and parameterized changing values in the command.

## Exercise 3: Add verbose output

### Scenario

In this exercise, you will modify your script so that it produces verbose output.

The main tasks for this exercise are as follows:

1. Add verbose output
2. Test the script

► **Task 1: Add verbose output**

1. Continuing on from the previous exercises, modify your Get-DiskInfo.ps1 script to produce verbose output when the script runs.
2. The verbose output should include the computer name and the drive type information that the script is using.

► **Task 2: Test the script**

1. In the Windows PowerShell ISE, switch to the Console pane.
2. Change to the **C:\** folder.
3. Run the script so that verbose output does not display.

4. Run the script so that verbose output is displayed.
5. Return the Script Pane.

**Results:** After completing this exercise, you will have changed your script so that it produces verbose output.

## Exercise 4: Add Comment-Based Help

### Scenario

In this exercise, you will add documentation to your script by using comment-based help.

The main tasks for this exercise are as follows:

1. Add comment-based help
2. Test the script

#### ► Task 1: Add comment-based help

1. Continuing on from the previous exercises, in your Get-DiskInfo.ps1 script, add comment-based help. Include at least the following items:
  - a. Synopsis
  - b. Description
  - c. **ComputerName** parameter
  - d. **DriveType** parameter
  - e. Example
2. Save the script.

#### ► Task 2: Test the script

1. In the Windows PowerShell ISE, switch to the Console pane.
2. Change to the C:\ folder.
3. Display the Help for Get-DiskInfo.ps1 in a floating window.
4. Return the Script Pane.

**Results:** After completing this exercise, you will have added documentation to your script by using comment-based help.

**Question:** What are some advantages of using comment-based help to document a script?

## Lesson 2

# Moving from Script to Function to Module

In this lesson, you will take a working, parameterized script and package it as a reusable tool. You will begin by wrapping the script's commands in a function. A function is one kind of Windows PowerShell command, and by creating one, you will be able to include multiple commands inside a single file. You will then structure that file as a script module that can be discovered and loaded by Windows PowerShell, and shared with colleagues or coworkers.

## Lesson Objectives

After completing this lesson, students will be able to:

- Transform a script into a function
- Describe the purpose and function of scope
- Transform a script into a script module
- Create debugging breakpoints in a script module

## Wrapping a Script in a Function

You currently have a script that performs a useful task. As you create more and more scripts, you may feel that you are creating lots of files that each have only a single use. Windows PowerShell has a way to combine several tools into a single file. This is known as a *script module*.

However, to combine several tools into one file, you must have a way to create a separation, or *wrapper*, around each tool. Doing this lets Windows PowerShell know which commands to run as a set. Windows PowerShell uses *functions* to provide a wrapper around a tool.

Basically, you make a function by wrapping all the contents of your script in a function declaration. A declaration consists of the keyword **function** followed by a function name. Function names should be consistent with the Windows PowerShell verb-noun naming convention.

For example, if you start with this:

- Wrap the body of your script, including comment-based help, in a function declaration

```
function Get-SecurityEvents {
    [CmdletBinding()]
    Param(
        [string]$ComputerName,
        [int]$EventID
    )
    Get-EventLog -LogName Security -ComputerName $ComputerName |
    Where EventID -eq $EventID |
    Select -First 50
}
```

```
[CmdletBinding()]
Param(
    [string]$ComputerName,
    [int]$EventID
)
Get-EventLog -LogName Security -ComputerName $ComputerName |
Where EventID -eq $EventID |
Select -First 50
```

The function might resemble this:

```
function Get-SecurityEvents {
```

```
[CmdletBinding()]
Param(
    [string]$ComputerName,
    [int]$EventID
)
Get-EventLog -LogName Security -ComputerName $ComputerName |
Where EventID -eq $EventID |
Select -First 50
}
```

Notice that the contents of the function are indented, helping make it visually obvious what commands belong to that function.

If your script includes comment-based help, include that help as the first thing inside the function. There are other areas where Windows PowerShell allows for comment-based help to be added, but adding the help immediately inside the beginning of the function is the easiest and has the fewest restrictions.

## Testing the Function

Unfortunately, testing the function is now somewhat more difficult. You cannot just run the script. Running the script defines the function to Windows PowerShell, but as soon as the script finishes, the function definition is deleted from memory. The function never actually runs. This behavior is because of a Windows PowerShell feature known as *scope*.

## Understanding Scope

*Scope* is a Windows PowerShell feature that defines containers around certain shell elements. The following items have their own scope:

- The shell itself has a scope called the *global scope*.
- Each script that you run is given its own scope.
- The inside of each function is given its own scope.

When an item finishes running, its scope is deleted from memory. Everything that an item creates is created in its own scope. If a script creates three new variables and two new functions, those five things are created in the script's scope. When the script finishes running, those five things are removed from memory together with that script's scope.

Scope is a complex topic, and complete coverage of scope is not included in this course. You can learn more about scope by running **help about\_scope -ShowWindow** in Windows PowerShell. For now, you have to remember that everything done inside a script can exist only until that script finishes running. That means that you have to make some changes to your script if you want to test your function.

- A scope is a container that the shell places around:
  - Itself
  - Each script you run
  - Each function you create
- The items created inside a scope, including variables and functions, only exist while that scope exists
- This behavior can make testing a function more challenging
- For more information, read **about\_scope**

## Demonstration: Testing the Function

In this demonstration, you will see how to test a function that is included inside a script.

### Demonstration Steps

1. Open the file **E:\Mod08\Democode\Function1.ps1**.
2. Add a command to run the function.

## Creating a Script Module

Your functions can be made easier to use by including them in a script module instead of including them in a script. When you save your module in the correct location, Windows PowerShell can find your module automatically, load it automatically, and make your functions available as commands inside the shell. You will no longer have to provide a path or file name to run your functions. Instead, you type their names, just as you would for any Windows PowerShell command.

- A script module can contain multiple functions
- When saved in the correct location, the shell can find and load the module automatically
- Functions in the script module behave like regular shell commands

A *script module* is any Windows PowerShell script that contains functions, and is saved in the correct location with the correct file name extension. The **PSModulePath** environment variable contains a list of correct locations. By default, it contains the following value:

```
C:\Users\<username>\Documents\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

You can view the variable on your computer by running **Get-Content Env:PSModulePath**.

 **Note:** One of the default module locations does not actually exist by default. The location in your Documents folder must be manually created before you can save modules there. You should not save modules in the System32 path. That location is reserved for modules created by Microsoft®.

If you have a script that you want to save as a script module, you have to decide on a unique module name. For example, **MyTools** is a possible module name. Module names should contain only letters and numbers, and no spaces or punctuation. If you selected **MyTools** as your module name, you would save the file as:

```
C:\Users\<username>\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1
```

Notice that the module must be saved with a **.psm1** file name extension, and that the file must be in a subfolder that has the module name. This is a very common point of confusion. For example, if the module name is MyTools, the file name must be MyTools.psm1 and it must be saved as \Modules\MyTools\MyTools.psm1, so that the file name and its containing folder name are the same.

 **Note:** Use caution when you save script modules in the Windows PowerShell ISE. If you just click **Documents** in the **Save As** dialog box, you are accessing the whole Documents Library. You should expand the library node and make sure that you have selected the Documents folder, instead of the **Public Documents** folder.

## Demonstration: Creating a Script Module

In this demonstration, you will see how to save a script as a script module.

### Demonstration Steps

1. Create the Windows PowerShell Modules folder.
2. Save the script as a script module.

## Adding Debugging Breakpoints

Windows PowerShell contains several features that help with debugging. You start with the least complex feature, the **Write-Debug** command.

**Write-Debug** works almost the same as **Write-Verbose**. By default, Windows PowerShell suppresses debugging output when you run a script or function. You must add the **-Debug** parameter to display the debug output. When **Write-Debug** displays output, it also pauses script execution. This pause gives you time to evaluate what the script has already done.

- Add **Write-Debug** to your script.
- Works much like **Write-Verbose**, and must be enabled by using the **-Debug** parameter.
- When enabled, displays your debug output and pauses the script. You can:
  - Continue execution
  - Halt execution
  - Suspend the script and examine the script environment from within the current scope

You also have the option to suspend the script.

When you suspend a script, Windows PowerShell displays a modified prompt. That prompt is basically inside the scope of whatever script or function you are running. You can view variables, run commands, and perform other tasks as if you are inside the script or function. When you are finished, run **Exit** to return to the debugging prompt.

The prompt also offers the option to cancel the script or function.

**Write-Debug** is not the most flexible or mature debugging technology in Windows PowerShell. However, it is one of the easier areas to start debugging. Many Windows PowerShell users add **Write-Debug** to their scripts as they are writing those scripts. Their **Write-Debug** commands serve as a kind of inline documentation, and by using them, users can start debugging immediately if they encounter a problem with their scripts.

## Demonstration: Adding and Using Debugging Breakpoints

In this demonstration, you will see how to add debugging breakpoints by using **Write-Debug**. You will also see how to use the suspend mode offered by the command.

### Demonstration Steps

1. Add a debug breakpoint to the script.
2. Save the script.
3. Switch to the Console pane.
4. Remove the **MyTools** module from memory.
5. Run the **Get-NetAdaptInfo** command. Specify **localhost** for the computer name. Do not enable debugging.

6. Run the **Get-NetAdaptInfo** command. Specify **localhost** for the computer name. Enable debugging.
7. At the debugging prompt, suspend the command.
8. Display the content of the **\$ComputerName** variable and verify that it contains **localhost**.
9. Exit suspend mode.
10. Let the command finish running.

**Question:** How could you set up your environment so that a set of script modules could be shared between yourself and your coworkers?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Moving from Script to Function to Module

### Scenario

You have written a script that performs an administrative task in your environment. You have to package that script as a Windows PowerShell script module so that it can be more easily used by other administrators in your environment.

### Objectives

After completing this lab, students will be able to:

- Convert a script to a function
- Convert a script to a script module
- Debug a script by using basic inline breakpoints

### Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click 10961B-LON-DC1, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. Perform the lab steps on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Convert the Script to a Function

#### Scenario

You have written the following script:

```
<#
.SYNOPSIS
Retrieves disk space information.
.DESCRIPTION
Retrieves disk information from a single computer.
```

```

.PARAMETER ComputerName
The name of the computer to query.
.PARAMETER DriveType
The type of drive to query. Defaults to 3, representing local fixed disks.
.EXAMPLE
.\Get-DiskInfo -ComputerName localhost -Verbose
#>
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)][string]$ComputerName,
    [int]$DriveType = 3
)
Write-Verbose "Getting drive types of $DriveType from $ComputerName"
Get-WMIObject -Class Win32_LogicalDisk -Filter "DriveType=$DriveType" ` 
    -ComputerName $ComputerName |
Select-Object -Property @{n='DriveLetter';e={$PSItem.DeviceID}}, 
    @{n='FreeSpace(MB)';e="{0:N2}" -f ($PSItem.FreeSpace / 
    1MB)}},
    @{n='Size(GB)';e="{0:N2}" -f ($PSItem.Size / 1GB)}},
    @{n='FreePercent';e="{0:N2}%" -f ($PSItem.FreeSpace / 
    $PSItem.Size * 100)}}

```

You now want to convert that script to a function, in preparation for packaging the script as a script module.

The main tasks for this exercise are as follows:

1. Add the function declaration
2. Execute the function
3. Test the script

► **Task 1: Add the function declaration**

1. In the Windows PowerShell ISE, open **E:\Mod08\Labfiles\LabB\Exercise1-Task1.ps1**.
2. Wrap all the contents of the script in a function named **Get-DiskInfo**.
3. Save the script as **C:\Tools.ps1**.

► **Task 2: Execute the function**

1. In your Tools.ps1 script, add the following to the end of the script:

```
Get-DiskInfo -Comp localhost
```

This command will execute the function when you run the script.

2. Save the script.

► **Task 3: Test the script**

- Run your **Tools.ps1** script.

**Results:** After completing this exercise, you will have converted the code in your script into a function.

## Exercise 2: Save the Script as a Script Module

### Scenario

You have written the following script:

```
function Get-DiskInfo {
    <#
    .SYNOPSIS
    Retrieves disk space information.
    .DESCRIPTION
    Retrieves disk information from a single computer.
    .PARAMETER ComputerName
    The name of the computer to query.
    .PARAMETER DriveType
    The type of drive to query. Defaults to 3, representing local fixed disks.
    .EXAMPLE
    .\Get-DiskInfo -ComputerName localhost -Verbose
    #>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)][string]$ComputerName,
        [int]$DriveType = 3
    )
    Write-Verbose "Getting drive types of $DriveType from $ComputerName"
    Get-WMIObject -Class Win32_LogicalDisk -Filter "DriveType=$DriveType" ` 
        -ComputerName $ComputerName |
    Select-Object -Property @{
        n='DriveLetter';e={$PSItem.DeviceID},
        n='FreeSpace(MB)';e={"{0:N2}" -f ($PSItem.FreeSpace / 1MB)},
        n='Size(GB)';e={"{0:N2}" -f ($PSItem.Size / 1GB)},
        n='FreePercent';
        e={"{0:N2}%" -f ($PSItem.FreeSpace / $PSItem.Size * 100)}
    }
    Get-DiskInfo -ComputerName localhost
```

You have to package this script as a Windows PowerShell script module.

The main tasks for this exercise are as follows:

1. Remove the function call
2. Save the script as a script module
3. Test the script module

► **Task 1: Remove the function call**

1. Continuing on from the last exercise in your C:\Tools.ps1 script, remove the line that runs the **Get-DiskInfo** function because it is no longer needed. This should be the last line in your script.
2. Save the script.

► **Task 2: Save the script as a script module**

- Save your Tools.ps1 script as a script module named **MyTools**.

► **Task 3: Test the script module**

1. In the Windows PowerShell ISE, switch to the Console pane.
2. Run the **Get-DiskInfo** command. Specify **localhost** as the computer name.
3. Remove the **MyTools** module from memory.

**Results:** After completing this exercise, you will have saved your script as a script module.

## Exercise 3: Add Debugging Breakpoints

### Scenario

You have created a script module named MyTools. In this exercise, you will add debugging breakpoints to the script module.

The main tasks for this exercise are as follows:

1. Add a breakpoint
2. Test the script module

#### ► Task 1: Add a breakpoint

1. Continuing on from the previous exercise in your MyTools.psm1 script, add a debugging breakpoint on the line after the **Write-Verbose** command. The debug output should include the computer name that the function is about to query.
2. Save the script module.

#### ► Task 2: Test the script module

1. In the Windows PowerShell ISE, switch to the Console pane.
2. Run the you **Get-DiskInfo** command. Specify **localhost** as the computer name, and do not enable debugging.
3. Run the **Get-DiskInfo** command. Specify **localhost** as the computer name, and enable debugging.
4. At the debug prompt, suspend the script.
5. Display the contents of **\$ComputerName**.
6. Exit debug mode.
7. Let the command continue.
8. Remove the **MyTools** module from memory.

**Results:** After completing this exercise, you will have added debugging breakpoints to the MyTools script module.

**Question:** If you have previous experience in scripting or programming, does **Write-Debug** work like other debuggers you have used?

## Lesson 3

# Implementing Basic Error Handling

In this lesson, you will learn to identify predictable errors in a script, and to add code and commands to gracefully handle those errors when they occur.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of error actions in Windows PowerShell
- Implement error handling constructs
- Implement error logging in a script

### Understanding Error Actions

When a PowerShell command has an error, that error may be one of two types. A *terminating* error is one where the command cannot possibly continue, and the command will always stop.

Terminating errors are less common. A *non-terminating* error is one where the command must stop its current action. However, it can continue to process other actions. For example, consider the following command:

```
Get-WmiObject -Class Win32_Service -  
ComputerName BAD,LON-DC1
```

- Terminating errors always make a command stop what it is doing
- A non-terminating error is one where a command can continue processing
- When a command has a non-terminating error, it needs to know what action you want it to take

If you assume that the computer BAD does not exist on the network, **Get-WmiObject** will have an error when it tries to query that computer. However, the command could potentially 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 has a non-terminating error, the command checks that variable to see what it should do. The variable can have one of four possible values:

- **Continue** is the default, and tells the command to display an error message and to 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.

For example:

```
$ErrorActionPreference = 'Inquire'
```

If you intend to trap an error within your script so that you can handle the error, commands must use the **Stop** action. You can trap and handle only terminating errors. However, it is considered a poor practice to

change **\$ErrorActionPreference** if the error that you expect will be produced by a Windows PowerShell command.

### -ErrorAction Parameter

All Windows PowerShell commands have an **-ErrorAction** parameter. This parameter has the alias **-EA**. The parameter accepts the same values as **\$ErrorActionPreference**, and the parameter overrides the variable for that particular command. If you expect an error occurring on a command, you therefore use **-ErrorAction** to set that command's error action to **Stop**. Doing this lets you trap and handle the error for that command, but leaves all other commands to use the action in **\$ErrorActionPreference**. For example:

```
Get-WmiObject -Class Win32_Service -ComputerName BAD,LON-DC1 -ErrorAction Stop
```

The only time that you will usually modify **\$ErrorActionPreference** is when you expect an error outside of a Windows PowerShell command, such as when you are executing a method:

```
Get-Process -Name Notepad | ForEach-Object { $PSItem.Kill() }
```

In this example, the **Kill()** method might have an error. But because it is not a Windows PowerShell command, it does not have an **-ErrorAction** parameter. You would instead set **\$ErrorActionPreference** to **Stop** before running the method, and then set the variable back to **Continue** after you run the method.



**Note:** You may find examples elsewhere that set **\$ErrorActionPreference** to **SilentlyContinue** at the beginning of a script. Be aware that this will suppress all error messages in the whole script. This would make debugging and troubleshooting very difficult. You should usually remove that setting before you run the script in your environment so that any errors will be displayed.

## Demonstration: Error Actions

In this demonstration, you will see all four error actions in operation.

### Demonstration Steps

1. Run a command by using the **Continue** error action.
2. Run a command by using the **SilentlyContinue** error action.
3. Run a command by using the **Inquire** error action.
4. Run a command by using the **Stop** error action.

## Try...Catch Constructs

Windows PowerShell uses the **Try...Catch...Finally** construct to implement error handling.

 **Note:** Windows PowerShell also uses the **Trap** construct for error handling. This construct was introduced in Windows PowerShell 1.0 and is more difficult and less flexible than **Try...Catch...Finally**. You should use **Try...Catch...Finally** in new scripts.

- **Try** section contains commands that might cause an error
  - Commands should process only one item
  - Set **-ErrorAction** to **Stop**
- **Catch** section runs if an error happens
- **Finally** section runs if an error happens or not

```
Try {  
    Get-WmiObject -Class Win32_Service -ComputerName $name -  
    ErrorAction Stop  
} Catch {  
    Write-Verbose "Error connecting to $name"  
}
```

As the name implies, the construct consists of up to three sections:

- The **Try** section contains commands that may have an error. These commands must have their **-ErrorAction** set to **Stop**, or the **\$ErrorActionPreference** variable must be set to **Stop**. This section is required.
- The **Catch** section is optional, and will run if an error happens in the **Try** section.
- The **Finally** section is optional, and will run regardless of whether an error happens in the **Try** section. If it is used, **Finally** must come after **Catch**.

You do not have to use both **Catch** and **Finally**. However, you must use at least one of them. You can have multiple **Catch** sections, configured so that each section handles a different kind of error. For more information about how to do this, read the **About\_Try\_Catch\_Finally** Help file in Windows PowerShell.

Here is an example:

```
Try {  
    Get-WmiObject -Class Win32_Service -ComputerName $name -ErrorAction Stop  
} Catch {  
    Write-Verbose "Error connecting to $name"  
}
```

 **Note:** The **Finally** section of this construct is rarely used. That is why the construct is known as **Try...Catch** instead of **Try...Catch...Finally**.

Remember that setting **-ErrorAction** to **Stop** will turn any error into a terminating error. That means the command will not continue to process. For this reason, you will usually write your script so that the command is processing only one thing at a time. In this example, that means **\$name** would contain only one computer name at a time, so if one computer name fails, the command is not skipping anything.

## Demonstration: Using Try...Catch

In this demonstration, you will see how to use **Try...Catch**.

### Demonstration Steps

- Add a command to a **Try...Catch** construct.

## Logging Errors

Although **-ErrorAction** tells a command what to do when an error happens, it does not capture the error. There are two ways to capture an error in Windows PowerShell.

The first is to use the built-in **\$Error** variable. This variable is an array, and the first item in the array is always the most recent error. The variable is global. This means that it can contain errors from the whole shell session. Accessing **\$Error[0]** provides the most recent error. This array will contain all errors, even those that resulted from an action other than running a Windows PowerShell command. For example, if you execute a method or an external application, and an error happens, **\$Error** will contain that error.

- Capturing an error allows you to log it, or to take different actions for different errors
- The built-in **\$Error** array stores the most recent error in **\$Error[0]**
  - Stores all errors throughout the shell
- Windows PowerShell commands use the **-ErrorVariable (-EV)** parameter to specify an alternate variable name for error storage
  - Variable names do not include a \$
  - Only stores errors for the specified command

When you run a Windows PowerShell command, a better approach is to use the **-ErrorVariable** parameter (which uses the alias **-EV**). The parameter accepts a variable name as its value, and if an error happens, the error is stored in that variable.

 **Note:** Variable names do not include a dollar sign (\$). Make sure not to include a dollar sign when specifying a variable name to **-ErrorVariable**.

For example:

```
Try {
    Get-WmiObject -Class Win32_Service -ComputerName $name -ErrorAction Stop -
    ErrorVariable MyErr
} Catch {
    Write-Verbose "Error connecting to $name"
    $MyErr | Out-File C:\Errors.txt
}
```

In this example, the error is captured to **\$MyErr** and written to the file C:\Errors.txt.

## Demonstration: Adding Error Logging

In this demonstration, you will see how to add error logging to a script.

### Demonstration Steps

- Use **-ErrorVariable** to add error logging to a script.

**Question:** The **Try...Catch** construct was introduced in Windows PowerShell 2.0. An earlier construct was introduced in Windows PowerShell 1.0, and is still supported in Windows PowerShell 2.0 and later. Do you know what that older construct is?

## Lab C: Implementing Basic Error Handling

### Scenario

You have written a Windows PowerShell script that may encounter errors when it is run in your environment. You have to change the script to handle and log those errors.

### Objectives

After completing this lab, students will be able to:

- Implement error handling in an existing function
- Test error handling capabilities in a function

### Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. Perform the lab steps on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Add Error Handling to a Function

#### Scenario

In this exercise, you will add error handling to an existing function that you created earlier in this module in Lab B.

The main tasks for this exercise are as follows:

1. Add error handling to a function
2. Test the function

► **Task 1: Add error handling to a function**

1. Continuing on from Lab B, open the **MyTools.psm1** script that you created in Lab B. Add error handling so that an error in the **Get-WMIObject** command displays verbose output that contains the name of the computer.
2. Save the script.

► **Task 2: Test the function**

1. Switch to the Console pane.
2. Remove the **MyTools** module from memory.
3. Run the **Get-DiskInfo** command. Specify **BAD** for the computer name, and enable verbose output.

**Results:** After completing this exercise, you will have added error handling to an existing function.

## Exercise 2: Add Error Handling to a New Function

### Scenario

In this exercise, you will add error handling to a function that someone else wrote.

The main tasks for this exercise are as follows:

1. Identify potential error points
2. Add error handling
3. Test the function

► **Task 1: Identify potential error points**

1. Open **E:\Mod08\Labfiles\LabC\Exercise2-Task1.ps1** in the Windows PowerShell ISE.
2. Identify the command that could cause an error if an incorrect computer name was specified.
3. Run the script and notice the behavior when an incorrect computer name is specified.

► **Task 2: Add error handling**

1. Add error handling to the **Exercise2-Task1.ps1** script. Your error handling must display a verbose message if an error happens. The message must include the name of the computer that could not be reached.
2. Save the script.

► **Task 3: Test the function**

- Run the script and notice the new behavior when an incorrect computer name is specified.

**Results:** After completing this exercise, you will have added error handling to a function that someone else wrote.

**Question:** In one of the demonstrations, you saw an example of how to add error logging to a script. That example used a static file name. In a production environment, a user might want the ability to specify the name of the file where errors would be logged. How could you provide that ability?

## Lesson 4

# Using Basic Scripting Constructs

In this lesson, you will learn about three of Windows PowerShell's scripting language constructs. These are the most frequently used elements of Windows PowerShell's scripting language.

### Lesson Objectives

After completing this lesson, students will be able to:

- Implement logical decision-making capabilities in scripts
- Implement choice-making capabilities in scripts
- Implement object enumeration loops in scripts

### The If Construct

The **If** construct is used to make logical decisions. Its complete syntax looks as follows:

```
If ($this -eq $that) {  
    # Condition 1  
} ElseIf ($those -gt $these) {  
    # Condition 2  
} ElseIf ($this -lt $that) {  
    # Condition 3  
} Else {  
    # Condition 4  
}
```

Notice that the **#** character in Windows PowerShell starts a comment line. The rest of the line is ignored by the shell.

```
If ($this -eq $that) {  
    # Condition 1  
} ElseIf ($those -gt $these) {  
    # Condition 2  
} ElseIf ($this -lt $that) {  
    # Condition 3  
} Else {  
    # Condition 4  
}
```

The **If** construct has three main components:

- The **If** section is mandatory. A condition that evaluates to either True or False must be enclosed in parentheses. Curly braces then enclose a script block. The script block contains the command or commands that will run if the condition evaluates to True.
- Zero or more **ElseIf** sections are next. Each of these has its own condition and script block. You are not required to have any **ElseIf** sections, and you may have as many as you must have.
- An optional **Else** section comes last. It does not have a condition. However, it does have a script block.

The conditions are evaluated in order. When a condition is found that evaluates to True, the shell executes the corresponding script block and stops evaluating any other conditions. If no conditions evaluate to True, and if an **Else** section exists, the **Else** section's script block will be executed.

As a best practice, the command or commands within each script block should be indented one level. You can create this indentation by using the Tab key in the Windows PowerShell ISE. The preceding formatting is one of two common ways to format this and other constructs. The following is the other common formatting approach:

```
If ($this -eq $that)  
{
```

```

    # Condition 1
}
Else
{
    # Condition 2
}

```

 **Note:** In the Windows PowerShell ISE, you can use the mouse pointer to select multiple lines of text. You can then press Tab or Shift+Tab to increase or decrease the indentation level for the selected lines.

## Demonstration: The If Construct

In this demonstration, you will see how to use the **If** construct.

### Demonstration Steps

1. Write a command that retrieves information about drive C.
2. Based on the value of the drive's **DriveType** property, display a message about the type of drive.

## The Switch Construct

The **Switch** construct is somewhat similar to an **If** construct that has multiple **ElseIf** sections. The **Switch** construct accepts a single value, or a variable that contains a value, or a property that contains a value. It compares that value to one of several possibilities that you provide. When it finds a match, it executes the corresponding script block. A **Default** section will execute only if no other matches were made. For example:

```

$drive = Get-CimInstance -ClassName Win32_LogicalDisk
$drive = Get-CimInstance -ClassName Win32_LogicalDisk -Filter
"DeviceID='C:'"
switch ($drive.DriveType) {
    3 { Write "Fixed local" }
    5 { Write "Optical" }
    Default { Write "Other" }
}

```

```

$drive = Get-CimInstance -ClassName Win32_LogicalDisk
switch ($drive.DriveType) {
    3 { Write "Fixed local" }
    5 { Write "Optical" }
    Default { Write "Other" }
}

• Also -Wildcard, -Regex, and other options
• Read About_Switch for more information
• Executes all matching conditions
• Use Break to stop matching

```

The **Switch** construct can also match by using wildcard patterns, regular expressions, and so on. To do this, you must specify a parameter that tells the construct what kind of match you want. For information about the available options, run **help about\_switch** in Windows PowerShell.

The construct differs from **If** in one important way. It will execute each matching script block, instead of executing only the first matching condition. For example:

```

$name = Get-Content Env:\COMPUTERNAME
switch -Wildcard ($name) {
    "LON_CL1*" {
        Write "Computer is a client"
    }
}

```

```
"LON*"    {
    Write "Computer is in LON"
}
"*DC*"    {
    Write "Computer is a DC"
}
"*1"      {
    Write "Computer is the first one"
}
Default   {
    Write "No matches"
}
}
```

In this example, LON-CL1 will display both **Computer is a client** and **Computer is the first one** because both conditions match. You can use the **Break** keyword to exit the construct. This prevents multiple matches. For example:

```
$name = Get-Content Env:\COMPUTERNAME
switch -Wildcard ($name) {
    "LON_CL1*" {
        Write "Computer is a client"
        break
    }
    "LON*"    {
        Write "Computer is in NYC"
    }
    "*DC*"    {
        Write "Computer is a DC"
        break
    }
    "*1"      {
        Write "Computer is the first one"
    }
    Default   {
        Write "No matches"
    }
}
```

LON-CL1 will now display only **Computer is a client**. The computer LON-DC1 would display **Computer is in NYC** and **Computer is a DC**.

## The **ForEach** Construct

The **ForEach** construct has the same purpose as the **ForEach-Object** cmdlet. That purpose is to enumerate through a collection of objects so that you can work with one object at a time. The construct has a different syntax than the cmdlet.



**Note:** The **ForEach-Object** cmdlet has an alias, **ForEach**, that is the same as the **ForEach** construct keyword. However, the cmdlet and the construct are not the same. Windows PowerShell decides whether you are using the cmdlet or the construct based on its position on the command

- Same purpose as **ForEach-Object** cmdlet, but different syntax

```
$ComputerNames = Get-Content Names.txt
ForEach ($name in $ComputerNames) {
    Write "The current name is $name"
}
```

line. Although both the construct and the cmdlet have the same purpose, they do not use the same syntax.

The construct looks as follows:

```
$ComputerNames = Get-Content Names.txt
ForEach ($name in $ComputerNames) {
    Write "The current name is $name"
}
```

In this example, Names.txt has one computer name per line. Those are loaded into **\$ComputerName** as individual String objects to make **\$ComputerName** a collection. The **ForEach** construct enumerates through those one at a time. The variable on the left side of the **in** keyword will contain one object at a time from the variable on the right side of the keyword. In other words, one computer name at a time will be taken out of **\$ComputerName** and added into **\$name**. The command or commands in the script block will execute one time for each object in the **\$ComputerName** variable.

Because most Windows PowerShell users prefer scripts that are easy to read and maintain, it is a common practice to use similar variable names in the **ForEach** construct. For example:

```
$Services = Get-Service
ForEach ($Service in $Services) {
    Write "The current service is $($service.name)"
}
```

However, this naming practice is not a technical requirement. The following example also works correctly:

```
$x = Get-Service
ForEach ($z in $x) {
    Write "The current service is $z"
}
```

This second example is not as easy to read, because the names **\$x** and **\$z** do not provide any indication or reminder of what the variables contain.

## Demonstration: The ForEach Construct

In this demonstration, you will see how to use the **ForEach** construct in a script.

### Demonstration Steps

- Use **ForEach** to enumerate through a list of computer names.

**Question:** Are you familiar with any scripting language constructs from other scripting languages that might also be present in Windows PowerShell?

## Lab D: Creating an Advanced Function

### Scenario

You are given a new administrative task that you have to automate in Windows PowerShell. You will automate this task by taking an existing command and turning it into an advanced function. You will include the advanced function in an existing script module.

### Objectives

After completing this lab, students will be able to:

- Test an existing command
- Convert a command to a parameterized function
- Modify a function to handle multiple targets
- Modify a function to include error handling

### Lab Setup

Estimated Time: 60 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. Perform the lab steps on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Test an Existing Command

#### Scenario

In this exercise, you will test an existing Windows PowerShell command to verify that it works correctly. You will also identify changeable values that you will have to parameterize later.

The main tasks for this exercise are as follows:

1. Test the command
2. Identify changeable values

► **Task 1: Test the command**

1. In the Windows PowerShell ISE, open **E:\Mod08\Labfiles\LabD\Exercise1.ps1**.
2. Run the script.

► **Task 2: Identify changeable values**

- In the Exercise1.ps1 script, identify one value that may have to change every time someone runs the script.

**Results:** After completing this exercise, you will have tested an existing command and identified changeable values.

## Exercise 2: Create a Parameterized Function

### Scenario

In this exercise, you will create a parameterized function by using the provided command.

The main tasks for this exercise are as follows:

1. Create the function
2. Add a parameter block
3. Parameterize the command
4. Test the function

► **Task 1: Create the function**

1. In the Windows PowerShell ISE, open your MyTools script module, which was created earlier in Lab C.
2. At the end of the MyTools.psm1 script module, create a new function named **Get-OSInfo**.
3. Copy and paste the contents of the Exercise1.ps1 script into the **Get-OSInfo** function.
4. Save the script.

► **Task 2: Add a parameter block**

1. Add a parameter block to the **Get-OSInfo** function in your MyTools.psm1 script module. The parameter block must include a single parameter and must also use the cmdlet binding attribute.
2. Save the script.

► **Task 3: Parameterize the command**

- In the **Get-OSInfo** function, change the value **localhost** to use your parameter name instead.

► **Task 4: Test the function**

1. Switch to the Console pane.
2. Remove the **MyTools** module from memory.
3. Run the **Get-OSInfo** command. Provide **localhost** for the computer name.
4. Remove the **MyTools** module from memory.

**Results:** After completing this exercise, you will have created a parameterized function by using the provided command.

## Exercise 3: Handle Multiple Targets

### Scenario

In this exercise, you will change a function to accept multiple computer names as input.

The main tasks for this exercise are as follows:

1. Modify a parameter
2. Implement an enumerating loop
3. Test the function

#### ► Task 1: Modify a parameter

1. In the **Get-OSInfo** function, modify the computer name parameter to accept multiple string values as input. To do this, change the parameter type to **[string[]]**.
2. Save the script.

#### ► Task 2: Implement an enumerating loop

1. In the **Get-OSInfo** function, add a loop that will enumerate through computer names one at a time.
2. Modify the command that uses the computer name to use the enumerator variable instead of the computer name parameter.
3. Save the script.

#### ► Task 3: Test the function

1. Switch to the Console pane.
2. Run the **Get-OSInfo** command. Provide **localhost,LON-DC1** for the computer name.
3. Remove the **MyTools** module from memory.

**Results:** After completing this exercise, you will have changed a function to accept multiple computer names as input.

## Exercise 4: Add error handling

### Scenario

In this exercise, you will add error handling to a function.

The main tasks for this exercise are as follows:

1. Add error handling
2. Test the function
3. To prepare for the next module

► **Task 1: Add error handling**

1. Add error handling to your **Get-OSInfo** function. Instead of displaying an error when an incorrect computer name is specified, the function should display a message that includes the incorrect computer name.
2. Save the script.

► **Task 2: Test the function**

1. Switch to the Console pane.
2. Run the **Get-OSInfo** command. For the computer name, specify **localhost,BAD,LON-CL1**

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have added error handling to a function.

**Question:** What if you need to write a function whose output combines information from multiple sources?

**Question:** What happens if you try to run Get-OSInfo and provide LON-CL1 as a target?

## Lesson 5

# Exploring Other Scripting Features

In this lesson, you will learn about some additional scripting-related capabilities of Windows PowerShell. This lesson is not meant to be a comprehensive examination of these features. Instead, this lesson is intended to give you a brief overview that provides a starting point for additional independent exploration.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the use of PSBreakpoints
- Describe additional features of advanced functions
- List additional scripting language constructs

### PSBreakpoints

PSBreakpoints are another, more advanced way to debug scripts. You can set these breakpoints visually in the ISE host application, or by running commands in either the console host or the ISE host. Breakpoints are contained within the Windows PowerShell session. Therefore, a breakpoint set in one Windows PowerShell window will not be effective in another Windows PowerShell window.

You can define breakpoints globally, or have them affect only a particular script. Breakpoints can be triggered when a specified command is run, when a script reaches a specified line number, or when a specified variable is read or changed. You can also define custom actions that run when a breakpoint is triggered.

- A more advanced, formal way of defining temporary breakpoints
- Can define breakpoints on:
  - Script line and column number
  - Command execution
  - Variable access
- Can specify custom actions for the breakpoint to execute, or allow the breakpoint to go into the debugging prompt
- For more, read **About\_Debuggers** in the shell

To see a list of commands that are available for working with breakpoints, run **Get-Command -Noun PSBreakpoint**. For a general overview of debugging when you use PSBreakpoints and other techniques, run **Help About\_Debuggers**.

## More Capabilities of Advanced Functions

*Advanced functions* are functions that have a **[CmdletBinding()]** declaration just before their **Param()** block. In addition to the capabilities that you have seen and used in this module, advanced functions have other capabilities:

- You can define parameter validation. Doing this lets the shell to validate parameter input according to your predefined rules.
- You can add support for the standardized – **WhatIf** and – **Confirm** parameters. This is a good technique when your function makes changes to the computer configuration, and it enables your function to follow standard Windows PowerShell practices.
- You can define aliases for parameter names. For example, you might define a – **HostName** alias for a – **ComputerName** parameter.

- Define validation for parameter input checking.
  - Add support for – **WhatIf** and – **Confirm**.
  - Define aliases for parameter names.
  - Create custom default formatting views.
  - Create custom type extensions.
  - And much more!
- Run **help \*function\*** in the shell to get a list of Help topics. Read those as a starting point for further exploration.

Advanced functions can also use any of the features that are available to regular functions and scripts. You can define customized default formatting views, type extensions, and use many different other techniques to make your functions work exactly the way that you want. These topics are beyond the scope of this course, but running **help \*function\*** in the shell will provide a list of Help files that can serve as a good starting point for independent exploration.

## Additional Scripting Constructs

The Windows PowerShell scripting language includes additional constructs that were not covered in this course. Some of these include the following:

- **For**, a construct that can repeat a block of code a specified number of times.
- **Do...While**, **While**, and **Do...Until**, three loops that can repeat a block of code until a specified condition is True or False.
- **Throw**, a keyword that allows your script or function to produce an exception
- **Break** and **Continue**, two keywords that can exit a loop or skip to the next iteration of a loop

- **For**
- **Do...While / While / Do...Until**
- **Throw**
- **Break**
- **Continue**

• Each is documented in an “About” Help file. Run **help about\*** for a complete list of these files.

Each of these is documented in a Help file within Windows PowerShell. For example, to learn more about the **For** construct, run **Help about\_For**.

**Question:** The main purpose for PSBreakpoints is to help debug scripts. You have also learned about the **Write-Debug** command, which serves a similar purpose. What other approaches to debugging are you familiar with?

## Module Review and Takeaways



**Best Practice:** As you start to write functions, take the time to format commands and code in the correct way. Every time that you begin a new construct, indent the contents of that construct. This technique helps make it visually clearer which code belongs to the construct to make both maintenance and debugging easier.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
A script module cannot be found by Windows PowerShell.	

### Review Question(s)

**Question:** What kinds of tasks do you want to automate immediately using a script module?

### Real-world Issues and Scenarios

As you write functions, remember that each function should produce as output only one kind of object. If you have to have a function to combine information from several sources, that information should be combined into a custom object that can hold all the information that you want. Creating custom objects is beyond the scope of this course. However, you can start by looking at the **New-Object** command in Windows PowerShell.

# Module 9

## Administering Remote Computers

### Contents:

Module Overview	09-1
<b>Lesson 1: Using Basic Remoting</b>	09-2
<b>Lesson 2: Using Advanced Remoting Techniques</b>	09-12
<b>Lab A: Using Basic Remoting</b>	09-15
<b>Lesson 3: Using Remoting Sessions</b>	09-18
<b>Lab B: Using Remoting Sessions</b>	09-23
Module Review and Takeaways	09-26

## Module Overview

Windows PowerShell™ remoting is a technology that enables you to connect to one or more remote computers and instruct them to run commands on your behalf. It is a powerful and flexible technology, and one that is quickly becoming the foundation for administrative communications in a Windows-based environment. For example, the graphical Server Manager console in Windows Server® 2012 relies on remoting to communicate with servers—even to communicate with the local computer on which the console is running! Past versions of the operating system have included remoting, but its use was largely optional. Now the technology is quickly becoming a mandatory component of every environment.

 **Additional Reading:** Detailed configuration of remoting is documented in a third-party free e-book titled "Secrets of PowerShell Remoting." You can find this book at [www.PowerShellBooks.com](http://www.PowerShellBooks.com).

 **Additional Reading:** The TechNet about\_Remote documentation can be found at <http://go.microsoft.com/fwlink/?LinkId=306155>.

### Objectives

After completing this module, students will be able to:

- Describe remoting architecture and security
- Manually enable remoting
- Use remoting for one-to-one and one-to-many connections
- Pass local variables to remote computers
- Create and manage persistent remoting sessions
- Use implicit remoting
- Use remoting to enable delegated administration

## Lesson 1

# Using Basic Remoting

Remoting is a complex technology in its own right, and to use it appropriately, you have to learn a bit about it. Actually working with remoting is fairly straightforward as soon as you understand the underlying concepts, and in this lesson you will begin to use remoting to perform administration on remote computers.

### Lesson Objectives

After completing this lesson, students will be able to:

- Describe the Windows PowerShell remoting architecture
- Explain the difference between remoting and other forms of remote administration
- Describe remoting security and privacy features
- Enable remoting on a computer
- Use remoting for single computer management
- Use remoting for multiple computer management
- 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 is used by web browsers to communicate with web servers. This makes the protocol easy to control and to route through firewalls. The Windows operating system implements the protocol in the Windows Remote Management (WinRM) service.

Remoting must be enabled on the computers that you want to receive incoming connections, although no configuration is necessary on computers that are initiating outgoing connections. The technology is enabled by default for incoming connections on Windows Server 2012 (and on later server operating systems), and can be enabled on any computer that is running Windows PowerShell 2.0 or later. Additionally, Windows PowerShell 2.0 and later can communicate with one another in a mixed-version environment.

- Protocol is WS-MAN, using HTTP (by default) or HTTPS
- Implemented by WinRM service
- Enabled by default on Windows Server 2012; available on any computer running PowerShell 2.0 or 3.0
- Must be enabled on any computer that will receive incoming connections
- Capable of supporting communications for a wide variety of applications, not just PowerShell



**Note:** Although remoting is enabled by default on Windows Server 2012 and later server operating systems, it is not enabled by default on client operating systems, including Windows 8 and Windows 8.1.

WinRM is used for Windows PowerShell remoting, and it is also able to handle communications for other applications. For example, on a default Windows Server 2012 (or later server operating system) installation, WinRM handles communications for 64-bit Windows PowerShell, 32-bit Windows PowerShell,

and two Server Manager components. In the future, it is likely that more and more applications, specifically on servers, will register with WinRM so that it will handle their administrative communications needs.

In this course, you will use remoting mainly in its default configuration, using HTTP on port 5985. Remoting can be configured to allow for or to require encryption based on Secure Sockets Layer (SSL), by using the HTTPS protocol, and defaults to port 5986 when doing this. However, to use HTTPS, a receiving computer must be configured to have an SSL certificate, which makes the remoting configuration somewhat more complex.

## Architecture

Remoting starts with the WinRM service. It registers one or more listeners. Each listener accepts incoming traffic through either HTTP or HTTPS, and each listener can be bound to a single local IP address or to multiple IP addresses. There is no dependency on Microsoft® Internet Information Services (IIS). This means that IIS does not have to be installed for WinRM to function.

Incoming traffic includes an envelope that indicates the traffic's intended destination, or *endpoint*. In Windows PowerShell, these endpoints are also known as *session configurations*. Each endpoint is associated with a specific application, and when traffic is directed to an endpoint, WinRM starts the associated application, hands off the incoming traffic, and waits for the application to complete its task. The application can pass data back to WinRM, and WinRM handles transmitting that data back to the originating computer.

In a Windows PowerShell scenario, you would send commands to WinRM, which would be executed by WinRM (the process is listed as Wsmprovhost in the remote computer's process list). Windows PowerShell would then execute those commands, and convert (or serialize) the resulting objects, if any, into XML. The XML text stream is handed back to WinRM, which transmits it to the originating computer. That computer's copy of Windows PowerShell deserializes the XML back into static objects. This enables the command results to behave much like any other objects within the Windows PowerShell pipeline.

Windows PowerShell can register multiple endpoints, or session configurations, with WinRM. In fact, a 64-bit operating system will register an endpoint for both the 64-bit Windows PowerShell host and the 32-bit host. This is by default. As you will see later in this module, you can also create your own custom endpoints that have highly precise permissions and capabilities assigned to them.

## Remoting vs. Remote Connectivity

Do not confuse *remoting*, the name of a specific Windows PowerShell feature, with the more generic concept of *remote connectivity*.

Many commands implement their own communications protocols, although in the future many of them may be changed to use remoting instead. For example, **Get-WmiObject** uses Remote Procedure Calls (RPCs), whereas **Get-Process** communicates with the computer's Remote Registry Service. Microsoft Exchange Server 2010 commands have their own communications channels, and Microsoft Active

Directory® commands talk to a specific web service gateway by using their own protocol. All these other forms of communication may have unique firewall requirements and may require specific configurations to be in place in order to operate.

- *Remoting* is the name of a specific feature that utilizes a specific service and protocol
- It applies to a relatively small subset of commands that can communicate with remote computers
- Just because a command has a – **ComputerName** parameter does not mean it uses remoting Nonremoting commands use their own protocols:
  - Remote Procedure Calls (RPCs), which include Windows Management Instrumentation (WMI)
  - Remote Registry Service (e.g., **Get-Process**)

*Remoting* is a generalized way to transmit *any* command to a remote computer for local execution. The command that you execute does not have to be present on the computer that initiates the connection; only the remote computers must be able to know about the command and to run it. The purpose of remoting was to reduce or eliminate the need for individual command authors to code their own communications protocols. Many command authors already had to do this to ship their products. This is why many different protocols and technologies are being used currently.

## Remoting Security

By default, the endpoints created by Windows PowerShell only allow for connections by members of a particular group. On Windows Server 2012 and Windows 8, this group is the Remote Management Users group, as well as the local Administrators group. On earlier operating system versions, members of the local Administrators group are allowed by default. In common practice, this means that remoting can be used only by domain administrators. Each endpoint does have a Security Access Control List (SACL) that can be changed to control exactly who can connect to it.

- Security transparent: you can do only what your credentials are allowed to do
- Auditing is not disabled or bypassed
- Mutual authentication helps prevent delegation of credentials to spoofed or impersonated computers
  - Works in domain environments by default
  - Can use SSL in lieu of domain credentials
  - Can be disabled through the TrustedHosts list

The default remoting behavior is to delegate your logon credentials to the remote computer, although you do have the option of specifying alternative credentials when you make a connection. Regardless, the remote computer uses those credentials to impersonate you, performing whatever tasks that you have specified by using those credentials. This behavior means that you will be able to do whatever *you* are allowed to do, and if those actions would usually be audited, they will be audited when they are performed through remoting also. Basically, remoting is *security transparent*, meaning it adds nothing, and removes nothing, from your environment's existing security. Whatever you could do when you were physically standing in front of the remote computer, you will be permitted to do through remoting, and nothing more.

## Security Risks and Mutual Authentication

Delegating your credential to a remote computer does carry some security risks. For example, if an attacker was able to successfully impersonate a known remote computer, you could potentially hand over highly privileged credentials to that attacker, who could then use them for malicious purposes. Because of this risk, remoting by default requires *mutual authentication*, which means that you must not only authenticate yourself to the remote computer, the remote computer must also authenticate itself to you. This guarantees that you connect only to the exact computer that you intended. This mutual authentication is a native feature of the Active Directory Kerberos authentication protocol, and when you connect between trusted domain computers, mutual authentication occurs automatically. When you connect to nondomain computers, you have to either provide another form of mutual authentication in the form of an SSL certificate (and the HTTPS protocol that must be set up in advance), or turn off the requirement for mutual authentication by adding the remote computer to your local TrustedHosts list.

## Computer Name Considerations

Because remoting must be able to look up a remote computer in Active Directory Domain Services, you must refer to computers only by their canonical computer name. IP addresses or DNS aliases, for example, will not work, because they do not provide remoting with the mutual authentication that remoting needs. If you *must* refer to a computer by IP address or by a DNS alias, either you must connect by using HTTPS

(meaning the remote computer must be configured to accept that protocol), or you must add the IP address or DNS alias to your local TrustedHosts list.

 **Note:** A special exception is made for the computer name **localhost**, which enables you to use it to connect to the local computer without any other configuration changes.

### The TrustedHosts List

The *TrustedHosts list* is a locally configured setting (that can also be configured by using Group Policy Object, or GPO) that lists the computers for which mutual authentication is not necessary. Computers must be listed with the same name that you will use to connect to them, whether that be an actual computer name, a DNS alias, or an IP address. Wildcards are permitted so that you could specify SRV\* to allow any computer whose name or DNS alias starts with "SRV." However, use caution with this list: although it does make connecting to nondomain computers without having to set up HTTPS easy, the TrustedHosts list *bypasses an important security measure*. It allows you to send your credential—potentially a very highly privileged one—to a remote computer without checking on whether that computer is in fact the one that you intended to connect to. The TrustedHosts list should be used only to designate computers that you know for a fact cannot be easily impersonated or compromised, such as servers housed in a protected data center. TrustedHosts might also be used to temporarily enable connections to nondomain computers on a controlled network subnet, such as new computers that are undergoing a provisioning process.

 **Best Practice:** Avoid using the TrustedHosts list unless absolutely necessary. Configuring a nondomain computer to use HTTPS is a more secure long-term solution.

### Privacy

By default, remoting uses HTTP, which does not offer privacy (encryption) for the contents of your communications. However, Windows PowerShell can and does apply application-level encryption by default. This means that your communications do receive a degree of privacy and protection. On internal networks, this application-level encryption is generally sufficient for most organizations' requirements.

Credentials are not typically transmitted in clear text.

In a domain environment that uses the default Kerberos authentication protocol, credentials are sent in the form of encrypted Kerberos tickets that do not include passwords.

When you connect by using HTTPS, the entire channel is encrypted by using the encryption keys of the remote computer's SSL certificate so that even if the Basic authentication protocol is used, passwords are not transmitted in the clear.

When you connect, by using HTTP and the Basic authentication protocol, to a computer that is not configured for HTTPS, credentials may be transmitted in clear text, including passwords.

The last situation would occur when you connect to a nondomain computer that you add to your local TrustedHosts list. Because credentials must be passed in clear-text in that scenario, you should ensure that you connect to a nondomain computer only on a controlled and protected network subnet, such as a subnet specifically designated for new computer provisioning. If you have to routinely connect to a nondomain computer, you should configure it to support HTTPS so that credentials are not transmitted in clear-text.

## Enabling Remoting

Be aware that remoting must be enabled only on computers that will *receive* incoming connections. No configuration is needed to enable *outgoing* communications (except to make sure that any local firewall will allow the outgoing traffic).

### Manually Enabling remoting

Run **Enable-PSRemoting** to manually enable remoting on a computer. This must be performed by a member of the local Administrators group, and is a persistent change (it can be disabled by later running **Disable-PSremoting**). This command will:

- Create an exception in the Windows Firewall for incoming TCP traffic on port 5985
- Create an HTTP listener on port 5985 for all local IP addresses
- Set the WinRM service to start automatically and restart it
- Register up to four default endpoints for use by Windows PowerShell

- Manually: Run **Enable-PSRemoting** as an Administrator
- Centrally: Configure a Group Policy Object (GPO)
- Note restrictions on client computers where a network connection is set to "Public"
- Remember that Windows Server 2012 enables Remoting by default; no further steps are needed

This command will fail on client computers where one or more network connections are set to Public (instead of Work or Home). You can override this failure by adding the **-SkipNetworkProfileCheck** parameter. However, be aware that the Windows Firewall will not allow exceptions when you are connected to a "Public" network.

### Enabling remoting by Using a GPO

Many organizations will prefer to centrally control remoting enablement and settings through GPOs. This is supported by Microsoft. You have to set up various settings in a GPO to duplicate the steps taken by **Enable-PSremoting**. The procedure is not difficult, however, it does involve several steps, and thus will not be covered in this course.

## Using Remoting: One-to-One

*One-to-one remoting* resembles the Secure Shell (SSH) tool that is used on many Unix and Linux computers in that you get a command prompt on the remote computer. The exact operation of remoting is very different from SSH. However, the end effect and usage is almost the same. In Windows PowerShell, you are technically typing commands on your local computer, which then transmits them to the remote computer for execution. Results are serialized into XML and transmitted back to your computer, which then deserializes them into objects and puts them into the Windows PowerShell pipeline. Unlike SSH, one-to-one remoting is not built on the Telnet protocol.

- Similar in concept to SSH, although different in actual operation
- Start with **Enter-PSSession -ComputerName name**
- Shell prompt changes to indicate connected computer
- Exit with **Exit-PSSession**

One-to-one remoting is engaged by using the **Enter-PSSession** command, combined with its **-ComputerName** parameter. Other parameters let you perform basic customization of the connection; these are covered later in this module.

When you are connected, the Windows PowerShell prompt changes to indicate the computer that you are connected to. Run **Exit-PSSession** to close the session and return to the local command prompt. If you close Windows PowerShell while connected, the connection will close on its own.

## Using Remoting: One-to-Many

*One-to-many remoting* lets you send a single command to multiple computers in parallel. Each computer will execute the command that you transmitted, serialize the results into XML, and transmit those results to your computer. Your computer deserializes the XML into objects, and puts them in the Windows PowerShell pipeline. When doing this, several properties are added to each object. This includes a **PSComputerName** property that indicates which computer each result came from. That property lets you sort, group, and filter based on computer name.

- **Invoke-Command** can send a command or script to one or more remote computers in parallel
- Results come back with a **PSComputerName** property indicating which computer each result came from
- Considerations include:
  - Throttling
  - Passing data to remote computers
  - Persistence
  - Ways to specify computer names

### Usage Scenarios

You can use one-to-many remoting in two main ways:

- **Invoke-Command -ComputerName *name1,name2* -ScriptBlock { *command* }**

This technique sends the command, or command contained in the script block, to the computers that are listed. This technique is useful for sending one or two commands (multiple commands are separated by a semicolon).

- **Invoke-Command -ComputerName *name1,name2* -FilePath *filepath***

This technique sends the contents of the designated script file (with a .ps1 file name extension) to the computers that are listed. The local computer opens the file and reads its contents. The remote computers do not have to have direct access to the file. This technique is useful for sending a large file of commands, such as a complete script.



**Note:** Within any script block, including the script block provided to the **-ScriptBlock** parameter, you can use a semicolon (;) to separate multiple commands. For example, **{ Get-Service ; Get-Process }** will run **Get-Service**, and then run **Get-Process**.

### Throttling

By default, Windows PowerShell will connect to only 32 computers at the same time. If you list more than 32 computers, the excess will be queued. As some initial computers complete and return their results, computers will be pulled from the queue and contacted.

You can alter this behavior by using the **-ThrottleLimit** parameter of **Invoke-Command**. Raising the number does not put additional load on the remote computers. However, it does put an additional load on the computer where **Invoke-Command** was run. Each concurrent connection is basically a thread of

Windows PowerShell. Therefore, raising the number consumes memory and processor on the local computer.

## Passing Values

The contents of the script block or file are transmitted as literal text to the remote computers that run them exactly as is. No parsing of the script block or file is performed by the computer where **Invoke-Command** was run. Consider the following example:

```
$var = 'BITS'  
Invoke-Command -ScriptBlock { Get-Service -Name $var } -Computer LON-DC1
```

In this scenario, the variable **\$var** is being set on the local computer. It is *not* inserted into the script block. In other words, LON-DC1 is being asked to retrieve a service whose name is equal to the one in the variable **\$var**. SERVER2, however, will have no idea what **\$var** contains, as it has not been defined on LON-DC1. This is a common mistake made by newcomers. There is a specific way to handle this situation, and it will be covered in the next lesson.

## Local Execution and Remote Execution

Pay close attention to the commands you enclose in the script block that will be passed to the remote computer. Remember that your local computer will not do anything with the contents of the script block. Those contents will be passed as is to the remote computer. For example, consider this command:

```
Invoke-Command -ScriptBlock { Do-Something -Credential (Get-Credential) }  
-ComputerName LON-DC1
```

This command will run the **Get-Credential** cmdlet on the remote computer. Try running **Get-Credential** on your local computer, and notice that it uses a graphical dialog box to prompt for the credential. Will that work when run on a remote computer? If you ran the preceding command on 100 remote computers, would you be prompted for 100 credentials?

Now consider this modified version of the command:

```
Invoke-Command -ScriptBlock { Param($c) Do-Something -Credential $c }  
-ComputerName LON-DC1  
-ArgumentList (Get-Credential)
```

This command runs **Get-Credential** on your local computer, and runs it only once. The resulting object is passed into the **\$c** parameter of the script block, enabling each computer to use the same credential.

These examples illustrate the importance of writing remoting commands carefully. By using a combination of remote execution and local execution, you can achieve a variety of useful goals.



**Note:** You will learn more about credential objects in Module 12, "Using Profiles and Advanced Windows PowerShell Techniques."

## Persistence

Using the technique outlined here, every time that you use **Invoke-Command**, the remote computer creates a new Windows PowerShell instance, runs your command or commands, returns the results to you, and then closes that Windows PowerShell instance. Each successive **Invoke-Command**, even if made to the same computers, is like opening a whole new Windows PowerShell window. Any work done by a previous session will not exist unless it was saved to disk or some other persistent storage. For example:

```
Invoke-Command -Computer LON-DC1 -ScriptBlock { $x = 'BITS' }  
Invoke-Command -Computer LON-DC1 -ScriptBlock { Get-Service -name $x }
```

In this example, the second **Invoke-Command** would fail, because it is dependant on a variable that was created in a previous instance of Windows PowerShell. When the first **Invoke-Command** finished running, that variable was lost. You can create a *persistent* instance of Windows PowerShell on a remote computer so that you can successfully send successive commands to it, and you will learn about that technique later in this module.

## Multiple Computer Names

The **-ComputerName** parameter of **Invoke-Command** can accept any collection of simple string objects as computer names. Here are many different techniques:

- **-ComputerName ONE,TWO,THREE**

This is a static, comma-separated list of computer names.

- **-ComputerName (Get-Content Names.txt)**

This reads names from a text file that is named Names.txt, assuming the file contains one computer name per line.

- **-ComputerName (Import-CSV Computers.csv | Select -Expand Computer)**

This reads a comma-separated values (CSV) file that is named Computers.csv. It contains a column named "**Computer**" that contains computer names.

- **-ComputerName (Get-ADComputer -filter \* | Select -Expand Name)**

This queries every computer object in Active Directory Domain Services (which can take significant time in a large domain).

## Common Mistakes When Using Computer Names

Be careful where you specify a computer name. For example:

```
Invoke-Command -ScriptBlock { Get-Service -ComputerName ONE,TWO }
```

This command does not provide a **-ComputerName** parameter to **Invoke-Command**. Therefore, the command runs on the local computer. The local computer will run **Get-Service**, and tell **Get-Service** to connect to computers named **ONE** and **TWO**. The protocols used by **Get-Service** will be used; Windows PowerShell remoting will not be used. Compare that with this 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.

## Demonstration: Enabling and Using Remoting

In this demonstration, you will see how to enable remoting on a client computer and how to use remoting in several basic scenarios.

### Demonstration Steps

1. Ensure you are signed into the 10961B-LON-CL1 virtual machine as Adatum\Administrator with password Pa\$\$w0rd.
2. Ensure you have the correct execution policy in place by running the command **Set-ExecutionPolicy RemoteSigned**.

3. Enable remoting.
4. Open a one-to-one connection to LON-DC1.
5. Get a list of processes running on LON-DC1.
6. Close the LON-DC1 connection.
7. Get a list of the most recent 10 Security event log entries from LON-CL1 and LON-DC1.

## Remoting Output vs. Local Output

When you run a command like **Get-Process** on your local computer, the command puts objects—in this case, objects of the type **System.Diagnostics.Process**—in the Windows PowerShell pipeline. These objects have properties, methods, and frequently events. Methods, you will recall, make the object do something. The **Kill()** method of a **Process** object, for example, ends the process that the object represents.

When a command runs on a remote computer, that computer serializes the results in XML, and transmits that XML text to your computer. You do this to put the object's information into a format that can be transmitted over a network. Unfortunately, the serialization procedure can deal only with static information about an object, in other words, its properties.

When the XML is received by your computer, the XML is deserialized back into objects that are put in the Windows PowerShell pipeline. When you have a **Process** object, piping it to **Get-Member** would reveal that it is now of the type **Deserialized.System.Diagnostics.Process**, a related, but different, kind of object. The deserialized object *has no methods and no events*.

So from a practical perspective, you should consider any data that is received through remoting to be a static snapshot. The data is not updatable, and the objects cannot be used to take any actions. Therefore, you will usually want to do as much processing as possible *on the remote computer*, where the objects are still live objects that have methods and events. For example:

```
Invoke-Command -Computer LON-DC1 -ScriptBlock { Get-Process -Name Note* } | Stop-Process
```

The preceding command would be a bad idea. You are receiving back **Process** objects; the action of stopping will occur on the local computer, not the remote one; and this action stops any local processes that happened to have names matching the remote ones. The correct approach would be as follows:

```
Invoke-Command -Computer LON-DC1 -ScriptBlock { Get-Process -Name Note* | Stop-Process }
```

Here, the processing has occurred completely on the remote computer, with only the final results being serialized and sent back. The difference between these two commands is subtle but important: make sure that you understand the difference.

**Question:** Why would an administrator decide to use remoting instead of managing a computer directly?

MCT USE ONLY  
STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

**Question:** What are some security concerns with remoting?

## Lesson 2

# Using Advanced Remoting Techniques

Remoting includes several advanced techniques that help achieve specific goals, or alleviate specific shortcomings. In this lesson, you will learn about the most useful advanced techniques.

### Lesson Objectives

After completing this lesson, students will be able to:

- Configure common remoting options
- Send parameters and local variables to remote computers
- Configure multihop remoting authentication

### Common Remoting Options

Both **Enter-PSSession** and **Invoke-Command** support several parameters that can let you change common connection options. These include the following:

- **-Port** is used to specify an alternative TCP port for the connection. Use this when the computer to which you are connecting is listening on a port other than the default 5985 (HTTP) or 5986 (HTTPS). You should be aware that you can, locally or through Group Policy, configure a different port as a permanent new default.
- **-UseSSL** instructs Windows PowerShell to use HTTPS instead of HTTP.
- **-Credential** specifies an alternative credential for the connection. This credential will be validated by the remote computer, and must have sufficient privileges and permissions to perform whatever tasks you intend to perform on the remote computer. This will be passed in clear-text when you are using a non-HTTPS connection and the Basic authentication protocol.
- **-ConfigurationName** connects to an endpoint (session configuration) other than the default. For example, specify “**Microsoft.Windows PowerShell32**” to connect to the remote computer’s 32-bit Windows PowerShell endpoint.
- **-Authentication** specifies an authentication protocol: the default is Kerberos, and other options include Basic, CredSSP, Digest, Negotiate, and NegotiateWithImplicitCredential. The protocol that you specify must be enabled in the WS-MAN configuration on both the initiating and receiving computers.

**--Port**  
**--UseSSL**  
**--Credential**  
**--ConfigurationName**  
**--Authentication**

Additional options available by creating a **PSSessionOption** object and passing it to **-SessionOption**

Additional session options can be configured by using **New-PSSessionOption** to create a new session option object, and passing it to the **-SessionOption** parameter of **Enter-PSSession** or **Invoke-Command**. Review the Help file for **New-PSSessionOption** to learn about its capabilities.

Defaults, such as the port number defaults and enabled authentication protocols, can be modified in the drive WSMAN in Windows PowerShell.

## Sending Parameters to Remote Computers

You have already learned that **Invoke-Command** cannot include variables in its script block or script file, unless those variables will be understood by the remote computer. It therefore may seem difficult to find a way to pass data from the initiating computer to the remote computers. However, the command actually provides a specific mechanism for doing this.

To review, the *intent* behind the following command is to display a list of the 10 most recent Security event log entries on each targeted computer. The command will not work as written.

- You can't just put local variables into the **Invoke-Command** script block
- You *can* pass data, however, you have to use a specific technique
- Pass local variables to the **-ArgumentList** parameter of **Invoke-Command**...
- ...and they'll map to variables in a **Param()** block *inside* the script block

```
$Log = 'Security'
$Quantity = 10
Invoke-Command -Computer ONE,TWO -ScriptBlock {
    Get-EventLog -LogName $Log -Newest $Quantity
}
```

The problem is that the variables **\$Log** and **\$Quantity** have meaning only on the local computer, and those values are not inserted into the script block prior to those values being sent to the remote computers. The remote computers are left to try to determine what they mean. Here's the correct syntax for this:

```
$Log = 'Security'
$Quantity = 10
Invoke-Command -Computer ONE,TWO -ScriptBlock {
    Param($x,$y) Get-EventLog -LogName $x -Newest $y
} -ArgumentList $Log,$Quantity
```

Here, the local variables are passed to the **ArgumentList** parameter of **Invoke-Command**. Within the script block, a **Param()** block is created. It contains the same number of variables as the **-ArgumentList** list of values, that is, two. The variables within the **Param()** block can be named anything that you want. They will receive data from the **ArgumentList** parameter based on order. In other words, because **\$Log** was listed first on **ArgumentList**, its value will be passed to **\$x** because it is first in the **Param()** block. The variables in the **Param()** block can then be used inside the script block, as shown.

 **Note:** The syntax shown in these examples will work for Windows PowerShell 2.0 and later. However, Windows PowerShell 3.0 introduced a simplified alternative syntax. If you have a local variable **\$variable**, and you want to include its contents in a command that will be executed on a remote computer, you can run **Invoke-Command -ScriptBlock { Do-Something \$Using:variable } -ComputerName REMOTE**. The special **\$Using:** prefix is understood by the local computer, and **\$Using:variable** will be replaced with the contents of the local variable **\$variable**.

This same technique works with the **-FilePath** parameter of **Invoke-Command**. In that case, Windows PowerShell expects the script file to already contain a **Param()** block, and will attach the **ArgumentList** values in the order in which they are listed.

## Demonstration: Sending Local Variables to a Remote Computer

In this demonstration, you will see the correct way to pass local information to a remote computer by using **Invoke-Command**.

### Demonstration Steps

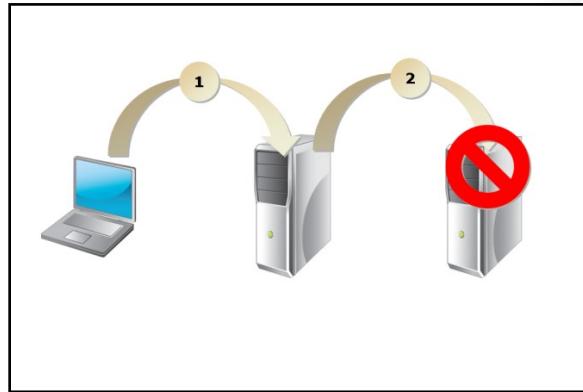
1. Create local variables to hold data.
2. Pass the local data to the remote computers to customize command execution.

## Multihop Remoting

One problem with remoting is how credentials are delegated. By default, credentials can be delegated across only one connection, or *hop*. This single delegation prevents the remote computer from further delegating your credentials, which might be a security risk.

However, the need to perform multiple hop, or *multihop*, delegation can be very real in production environments. For example, in some organizations, administrators are not permitted to connect directly from their client computers to a server in the data center. Instead, they must connect to an intermediate gateway or jump server, and then from there connect to the server they intend to manage. In its default configuration, remoting does not permit this. After you are connected to a remote computer, your credential can go no further. Trying to access any resource not located on that computer typically results in a failure, because your access is not accompanied by a credential.

The solution is to enable Credential Security Support Provider (CredSSP), a new authentication protocol introduced with the Windows Vista® operating system and present in Windows 7 and Windows 8.



### Enabling CredSSP

The CredSSP protocol must be enabled 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 **Enable-WsManCredSSP –Role Client –Delegate *servername***. Substitute *servername* with the name of the server that will be able to redelegate your credential. The server name can contain wildcard characters, although using only \* is too permissive, because you would be enabling any computer, even an intruder's, to redelegate your credential. Instead, consider a limited wildcard pattern, such as \*.ADATUM.com, that would limit redelegation to computers in that domain.

To configure the server, run **Enable-WsManCredSSP –Role Server**. No delegated computer list is needed on the server.

These settings can also be configured through Group Policy, offering a more centralized and consistent configuration across an enterprise.

**Question:** Why might you configure remoting to use ports other than the defaults?

# Lab A: Using Basic Remoting

## Scenario

You are an administrator for ADATUM, and must perform some maintenance tasks on a server. You do not have physical access to the server, and instead plan to perform the maintenance tasks by using Windows PowerShell remoting. The server in question runs Windows Server 2012. You also have some tasks that must be performed against both a server and another client computer that runs Windows 8.

## Objectives

After completing this lab, students will be able to:

- Enable remoting on a client computer
- Execute a task on a remote computer by using one-to-one remoting
- Execute a task on two computers by using one-to-many remoting

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. Perform the lab steps on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Enabling Remoting on the Local Computer

### Scenario

In this exercise, you will enable remoting on the client computer.

The main tasks for this exercise are as follows:

1. Enable remoting for incoming connections

► **Task 1: Enable remoting for incoming connections**

1. Ensure you are signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd** with the Windows PowerShell console open.
2. Ensure the **ExecutionPolicy** is set to **RemoteSigned**.
3. Enable remoting for incoming connections on the **LON-CL1** client computer.
4. Verify that Windows PowerShell has created several session configurations. You have to find a command that can produce this list.

**Results:** After completing this exercise, you will have enabled remoting on the client computer.

## Exercise 2: Performing One-to-One Remoting

### Scenario

In this exercise, you will connect to a remote computer and perform maintenance tasks.

The main tasks for this exercise are as follows:

1. Connect to the remote computer and install an operating system feature on it.
2. Test multihop remoting
3. Observe remoting limitations

► **Task 1: Connect to the remote computer and install an operating system feature on it.**

1. Ensure you are still signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. On **LON-CL1**, Use remoting to establish a one-to-one connection to LON-DC1.
3. Install the Network Load Balancing (NLB) feature on LON-DC1.
4. Disconnect from LON-DC1.

► **Task 2: Test multihop remoting**

1. Establish a one-to-one remoting connection with LON-DC1.
2. Try to establish a connection from LON-DC1 to LON-CL1. What happens, and why?
3. Close the connection.

► **Task 3: Observe remoting limitations**

1. Ensure you are signed in to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Establish a one-to-one remoting connection to LON-CL1 (use the computer name **localhost**). This is your local computer, but this new connection creates a second user session for you on the computer.
3. Use Windows PowerShell to start a new instance of Notepad. What happens? Why?
4. Stop the pipeline to exit the Notepad process.
5. Close the connection.

**Results:** After completing this exercise, you will have connected to a remote computer and performed maintenance tasks on it.

## Exercise 3: Performing One-to-Many Remoting

### Scenario

In this exercise, you will run commands against multiple computers. One of those will be the client computer, although you will be establishing a second logon to it for the duration of each command.

The main tasks for this exercise are as follows:

1. Retrieve a list of physical network adapters from two computers.
2. Compare the output of a local command to that of a remote command.

► **Task 1: Retrieve a list of physical network adapters from two computers.**

1. Ensure you are still signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. On **LON-CL1**, Using a keyword like **adapter**, find a command that can list network adapters.
3. Read the Help for the command and find a switch parameter that will limit output to physical adapters.
4. Use remoting to run the command on LON-CL1 and LON-DC1.

► **Task 2: Compare the output of a local command to that of a remote command.**

1. Pipe a collection of **Process** objects to **Get-Member**.
2. Use remoting to retrieve a list of **Process** objects from LON-DC1, and pipe them to **Get-Member**.
3. Compare the output of the two **Get-Member** results.

**Results:** After completing this exercise, you will have run commands against multiple remote computers.

**Question:** Would it be possible to use remoting to connect to a client computer and run an application that the interactive user of that computer could see?

## Lesson 3

# Using Remoting Sessions

To this point, you have worked with remoting only in an ad hoc manner. That is, each remoting command that you used created a connection, used it, and then closed it. You have already learned that this model does not offer any kind of persistence of information on the remote computer. In this lesson, you will learn how to establish and manage *persistent* connections to remote computers, known as *sessions*.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of remoting sessions
- Create a remoting session
- Transmit commands by using a remoting session
- Disconnect and reconnect remoting sessions
- Execute remote commands by using implicit remoting

### Persistent Connections

To this point, when you have run **Enter-PSSession** or **Invoke-Command**, Windows PowerShell has had to make the connection to the remote computer, retrieve a copy of Windows PowerShell running there, run whatever commands that you specified, and then close the remote copy of Windows PowerShell and close the connection. This ad hoc technique offers no persistence of information across connections, because each connection is basically starting a brand-new copy of Windows PowerShell.

Windows PowerShell does have the capability of creating persistent connections that are known as sessions, or more accurately, *PSSessions* (the "PS" designation signifies Windows PowerShell, and differentiates these sessions from other kinds of sessions that might be present in other technologies, such as a Remote Desktop Services session).

- Sessions—or, technically, *PSSessions*
- Represent a persistently running remote copy of PowerShell
- Can execute multiple sequences of commands, be disconnected and reconnected, and closed
- Numerous configuration parameters in the drive WSMancontrol idle session time, maximum connections, etc.

### How Sessions Work

You start by creating a session that establishes a connection to a remote computer and starts a copy of Windows PowerShell there, which is the exact process that you have used in remoting to this point. You run whatever commands you want within the session, but then you leave it running. The session can be targeted for additional commands, all of which can create and use persistent information in the remote copy of Windows PowerShell. When you are finished, you close the session, closing the remote copy of Windows PowerShell.

## Disconnected Sessions

In Windows PowerShell 3.0 and later, sessions can also be manually disconnected, closing your connection but leaving the remote copy of Windows PowerShell running. You can then reconnect to the session, even from a different computer, to continue using that same copy of Windows PowerShell.

## Controlling Sessions

Every computer where remoting is enabled has a drive WSMAN that includes many configuration parameters related to session. This includes maximum session run time, maximum idle time, maximum number of incoming connections, and maximum number of sessions per administrator. You can explore these by running **Dir WSMAN:\localhost\shell**, and can change them in that same location. Many of the settings can also be controlled through Group Policy.

## Creating a Session

Use **New-PSSession** to create a new session. You will notice that the command contains many of the same parameters as **Invoke-Command**, including, among others, **Credential**, **-Port**, and **-UseSSL**. You are creating the identical kind of connection that **Invoke-Command** creates you are just leaving this connection running, instead of immediately closing and using it.

Sessions do have an idle time-out, after which they will be closed by the remote computer. A closed session differs from a disconnected one, and closed sessions cannot be reconnected. All that you can do is remove them and create a brand-new one.

- Create sessions by using **New-PSSession**
- The command produces a reference to the session(s) it created
- Assign session(s) to variable(s) to make them easier to refer to

**New-PSSession** can accept multiple computer names. This causes it to create multiple session objects. When you run the command, it outputs objects representing the newly created sessions. You will frequently assign these to a variable to make them easier to refer to and use in the future. For example:

```
$dc = New-PSSession -ComputerName LON-DC1
```

## Using a Session

As soon as you have created a session, you can use it. Both **Invoke-Command** and **Enter-PSSession** can accept a session object (**Invoke-Command** can accept multiple session objects) instead of a computer name. Use the commands' **-Session** parameter for this purpose. When you do, they use the existing session instead of creating a new connection. When your command finishes running or you exit the session, the session *remains running and connected* and ready for future use. For example:

- Pass a session object to the **-Session** parameter of **Enter-PSSession** to interactively enter that session
- Or, pass 1+ session object to the **-Session** parameter of **Invoke-Command** to run a command against those sessions
- The sessions remain open and connected after you are finished, leaving them ready for additional use

```
$client = New-PSSession -ComputerName
```

```
LON-CL1  
Enter-PSSession -Session $client  
Exit-PSSession
```

Or:

```
$computers = New-PSSession -ComputerName LON-CL1,LON-DC1  
Invoke-Command -Session $computers -ScriptBlock { Get-Process }
```

## Demonstration: Using Sessions

In this demonstration, you will see how to create and manage sessions.

### Demonstration Steps

1. On the LON-CL1 virtual machine, create a session to LON-DC1 and store it in a variable.
2. Create sessions to LON-CL1 and LON-DC1 and store them both in a single variable.
3. Display a list of all open sessions.
4. Display the status of the LON-DC1 session.
5. Interactively enter the LON-DC1 session.
6. Display a list of running processes.
7. Exit the session.
8. Display the status of the LON-DC1 session.
9. Use remoting to retrieve a list of started services from both LON-CL1 and LON-DC1, by using the already-open sessions.
10. Close the session to LON-DC1.
11. Display a list of all open sessions.
12. Close all open sessions.

## Disconnected Sessions

As you have learned, Windows PowerShell can disconnect sessions 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 **Disconnected** state if the connection is interrupted. However, if you manually close the Windows PowerShell host application, it will not disconnect sessions but will instead close them. Using disconnected sessions resembles the following process:

- **Disconnect-PSSession** to disconnect a session while leaving PowerShell running
  - Does not happen automatically when you close the host application
- **Get-PSSession -ComputerName** displays a list of your sessions on the specified computer
  - You cannot see other users' sessions
- **Connect-PSSession** reconnects a session, making it available for use

1. Use **New-PSSession** to create the new session. Optionally, use the session to run commands.

2. Run **Disconnect-PSSession** to disconnect the session. Pass the session object that you want to disconnect 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 sessions running on the specified computer.
5. Use **Connect-PSSession** to reconnect the desired session.



**Note:** You cannot see or reconnect another user's sessions on a computer.

## Demonstration: Disconnected Sessions

In this demonstration, you will see how to use disconnected sessions.

### Demonstration Steps

1. Create a session to LON-DC1. Save the session in a variable.
2. Disconnect the session.
3. List the sessions open in LON-DC1.
4. Reconnect the disconnected session.
5. Verify that your variable contains an active and usable session.
6. Close and remove the session.

## Implicit Remoting

Another good use of sessions is called *implicit remoting*.

One of the ongoing problems in the Windows management space is *version mismatch*. For example, Windows Server 2012 includes several great Windows PowerShell commands. Those commands can be installed on Windows 8 as part of the Remote Server Administration Tools (RSAT). But the Windows 8 RSAT cannot be installed on Windows 7—if your workstation runs Windows 7, you would seem to be out of luck.

- Imports commands from a remote computer to the local one
- Imported commands still run on the remote computer, through an established Remoting session
- Lets you utilize commands without needing to install them
- Help is also drawn from the remote computer

If you have recently had to rebuild your workstation, you are familiar with another ongoing problem. The sheer amount of time that it can take to track down and install administrative tools and consoles on a computer. Even assuming all of them are compatible with your version of Windows, just installing them can take days.

These problems are why so many administrators forgo installing any tools on their workstations, and access tools directly on the server through Remote Desktop. This, unfortunately, is a terrible solution, because it puts the server in the position of having to be a client, while at the same time providing services to hundreds or thousands of users. The advent of Server Core, which lacks a graphical user

MCT USE ONLY. STUDENT USE PROHIBITED

interface, was in part to make servers perform better and need fewer updates—but that also means that they cannot run graphical tools and consoles.

So what do you do?

### Implicit Remoting Brings Tools to You

The purpose of *implicit remoting* is to bring a copy of a server's Windows PowerShell tools to your local computer. In reality, you are not copying the commands at all; you are just creating a kind of shortcut, called a *proxy function*, to the server's commands. When you run the commands on your local computer, they are implicitly run on the server through remoting. Results are sent back to you. It is exactly as if you ran everything through **Invoke-Command** but much more convenient. Commands also run more quickly, because commands on the server are naturally co-located with the server's functionality and data.

### Using Implicit Remoting

This feature was available in Windows PowerShell version 2.0, but is much easier to use in version 3 and later. Just create a session to the server that contains the module that you have to use. Then, use **Import-Module** and its **-PSSession** parameter to import the desired module. The commands in that module, and even its Help files, become available in your local Windows PowerShell console during that session.

You have the option of adding a prefix to the noun of the commands that you import in this manner. Doing this can make it easier, for example, to have multiple versions of the same commands loaded at the same time, without causing a naming collision. For example, you might import both Microsoft Exchange Server 2010 and Microsoft Exchange Server 2013 commands, adding a **2010** and **2013** prefix, respectively. This enables you to run both sets of commands. In reality, each would be running on their respective servers, enabling you to run both sets (perhaps in a migration scenario) side-by-side.

Help also works for commands that are running through implicit remoting. However, the Help files are drawn through the same remoting session as the commands themselves. Therefore, the remote computer must have an updated copy of its Help files. This can be a concern on servers, because they may not be used all that frequently, and nobody may have run **Update-Help** on them to pull down the latest Help files.

### Demonstration: Implicit Remoting

In this demonstration, you will see how to use implicit remoting to import and use a module from a remote computer.

#### Demonstration Steps

1. On 10961B-LON-CL1, establish a remoting session to LON-DC1 and save it in a variable.
2. Get a list of modules on LON-DC1.
3. Import the ActiveDirectory module from LON-DC1, adding the prefix **Rem** to the imported commands' nouns.
4. Display Help for **Get-RemADUser**.
5. Display a list of all domain users.
6. Close the connection to LON-DC1.
7. Try running **Get-RemADUser**.

**Question:** What are some potential operational concerns for sessions?

# Lab B: Using Remoting Sessions

## Scenario

You are an administrator who must perform multiple management tasks against remote computers. In your environment, communications protocols like RPCs are blocked between you and the servers. You plan to use Windows PowerShell remoting, and want to use sessions to provide persistence and to reduce the setup and cleanup overhead imposed by ad hoc remoting connections.

## Objectives

After completing this lab, students will be able to:

- Create and manage remoting sessions
- Import commands from a remote computer
- Send commands to multiple computers in parallel

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V Manager, click 10961B-LON-DC1, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Using Implicit Remoting

### Scenario

In this exercise, you will use implicit remoting to import and run commands from a remote computer.

The main tasks for this exercise are as follows:

1. Create a persistent remoting connection to a server
2. Import and use a module from a server
3. Close all open remoting connections

► **Task 1: Create a persistent remoting connection to a server**

1. On **10961B-LON-CL1**, logged in as **Adatum\Administrator** with password **Pa\$\$w0rd**, in the Windows PowerShell console, create a persistent remoting connection to LON-DC1, and save the resulting session object in the variable **\$dc**.
2. Display a list of sessions in **\$dc** and verify their availability.

► **Task 2: Import and use a module from a server**

1. Display a list of Windows PowerShell modules on LON-DC1.
2. Locate a module on LON-DC1 that can work with Server Message Blocks (SMB) shares.
3. Import the module to your local computer, adding the prefix **DC** to the nouns for each imported command.
4. Display a list of SMB shares on LON-DC1.
5. Display a list of SMB shares on the client computer.

► **Task 3: Close all open remoting connections**

- Close all open remoting connections.

**Results:** After completing this exercise, you will have used implicit remoting to import and run commands from a remote computer.

## Exercise 2: Multicomputer Management

### Scenario

In this exercise, you will perform several management tasks against multiple computers, relying on remoting sessions to provide persistence.

The main tasks for this exercise are as follows:

1. Create remoting sessions to two computers
2. Create a report that displays Windows Firewall rules from two computers
3. Create and display an HTML report that displays local disk information from two computers. Your report must include each computer's name, each drive's letter, and each drive's free space and total size in bytes.
4. Close and remove all open remoting sessions
5. To prepare for the next module

► **Task 1: Create remoting sessions to two computers**

1. Ensure you're still signed in to **10961B-LON-CL1** as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Create remoting sessions to LON-CL1 and LON-DC1 and save both session objects in the variable **\$computers**.
3. Verify that the connections in **\$computers** are open and available.

► **Task 2: Create a report that displays Windows Firewall rules from two computers**

1. Discover a Windows PowerShell module capable of working with Network Security.

2. Use a single command line to load the module into memory on LON-CL1 and on LON-DC1.
3. Discover a command that can display Windows Firewall rules.
4. Use a single command line to list all enabled firewall rules on LON-CL1 and LON-DC1. Display only the rule names and the computer name each rule came from.
5. Use a single command line to unload the network security module from memory on LON-CL1 and LON-DC1.

► **Task 3: Create and display an HTML report that displays local disk information from two computers. Your report must include each computer's name, each drive's letter, and each drive's free space and total size in bytes.**

1. As a test, use **Get-WmiObject** to display a list of local hard drives (the **Win32\_LogicalDisk** class, filtered to include only those drives with a drive type of 3).
2. Use remoting to run the **Get-WmiObject** command against LON-DC1 and LON-CL1. Do not add a **-ComputerName** parameter to the **Get-WmiObject** command.
3. Revise your command from the previous step to produce the desired HTML report.

► **Task 4: Close and remove all open remoting sessions**

- Remove all open remoting sessions.

► **Task 5: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have performed several management tasks against multiple computers.

**Question:** What are some benefits offered by implicit remoting?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways



**Best Practice:** Always consider the security implications of opening remoting too much. The default configuration is tightly locked down and provides a good balance of ease-of-use and security/privacy; make sure that, before changing that default configuration, you have explored all the possible ramifications.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
General problems using remoting.	
Remoting will not enable on a client computer.	

### Real-world Issues and Scenarios

Many organizations express concerns about remoting's security, and frequently decide to disable it. Remoting is just as secure as Remote Desktop Protocol, and in most ways is more secure and more controllable than the many protocols (such as RPCs) that it tries to replace. As of Windows Server 2012, remoting is required: native tools such as Server Manager rely on remoting to configure even the local computer. IT security personnel should take the time to thoroughly understand remoting before just deciding to disable it, as disabling it deprives administrators of a valuable management tool, frequently requiring them to take less-secure workaround measures to restore lost functionality.

There is also misplaced concern over the CredSSP protocol, that Microsoft describes as an increased security risk because it enables the delegation of credentials to remote computers, and if those computers are compromised then the credential could also be compromised. The key phrase is "if those computers are compromised," reinforcing the fact that only trusted, managed, secured computers should be enabled for CredSSP delegation. This concern is why delegating to a wildcard character such as \* is incorrect—doing this would allow delegation to any computer. This includes untrusted, unmanaged, unsecured computers—and that is where the security risk lies.

# Module 10

## Putting it All Together

### Contents:

Module Overview	10-1
<b>Lesson 1:</b> Provisioning a New Server Core Instance	10-2
<b>Lab:</b> Provisioning a New Server Core Instance	10-6
Module Review and Takeaways	10-14

## Module Overview

In this module, you will complete a real-world task from scratch: performing the initial provisioning of a newly installed Microsoft Windows Server® 2012 Server Core instance. You will break this task down into several discrete subtasks, and perform each one individually at the command prompt. After completing each task at the command prompt, you will add that task's commands to a script, creating an automated provisioning process.



**Additional Reading:** Although you do not need to know any details about Server Core to complete this module, you can learn more about it at the Server Core Installation Option Getting Started Guide page, <http://go.microsoft.com/fwlink/?LinkId=306156>.

### Objectives

In the module, you will:

- Discover, test, and execute new commands that help complete individual tasks in a larger overall process

## Lesson 1

# Provisioning a New Server Core Instance

In this lesson, you will review the process that you must complete, and break the process down into individual tasks. Your instructor will briefly review these tasks but will not provide much guidance. You should refer back to the lesson content as you work on the lab if you want reminders or reference materials.

### Lesson Objectives

After completing this lesson, students will be able to:

- Review the overall process that you will automate
- Create an initial parameterized script
- Describe the requirements and criteria of the process that you will automate

### Overall Process

The overall process that you will automate is the core provisioning of a Server Core computer that is running Microsoft® Windows Server® 2012. The computer has the operating system installed, and a local Administrator password is set. Everything else must be automated by a script that you will write.

- Provision a new Windows Server 2012 computer installed as Server Core
- A local Administrator password has been set
- You will need to write a script to perform the remainder of the provisioning process in accordance with your organization's guidelines

### Create a Parameterized Script

- Parameterize values that will change each time a new server is provisioned
- Leave static any values that rarely or never change in the environment
- What are some examples of each kind of value?

To make your script more easily reused, you will write it as a parameterized script. Certain values within the script will remain hard-coded, such as those values that will rarely change. Those values include the name of a DHCP server, your domain name, and other values that are fairly static in any environment.

You will parameterize the values that will change every time that a new server is provisioned. Those values may include the following:

- The computer's new name
- The computer's new IP address
- The computer's physical network adapter media access control (MAC) address
- The role or roles to install on the computer
- The credentials that are used to access the computer and the domain

In a production environment, you might perform additional tasks when provisioning a new server, and you might have to parameterize some of that information.

You should be aware that, in a well-designed environment, many secondary provisioning tasks would be performed by another method. For example, software would be deployed through Microsoft System Center 2012 Configuration Manager or by Group Policy Objects (GPOs). Configuration settings would be pushed out through Group Policy objects. Other provisioning tasks might be performed by other infrastructure components. For this reason, this module will focus only on the initial provisioning tasks that cannot be performed by these other common methodologies and technologies.

## Review Criteria and Tasks

### Retrieve the Current Dynamic IP Address

Your first step will be to retrieve the new computer's current IP address. By default, Windows computers use Dynamic Host Configuration Protocol (DHCP) to obtain network configuration information, and this is the case with the new Server Core computer. The one piece of information that you could typically expect to have is the MAC address assigned to the computer's network adapter. On a physical computer, this is frequently printed on a label on the computer's case; for a virtual machine, hypervisor configuration tools can frequently provide this information.

- Retrieve the current dynamic IP address
- Add a reservation to the DHCP scope
- Modify the local TrustedHosts list
- Add the server to the domain
- Add a role to the server
- Restore the TrustedHosts list

Your lab environment includes a Microsoft DHCP Server on the LON-DC1 computer. Your LON-CL1 client computer includes the Remote Server Administration Tools (RSAT) and includes Windows PowerShell™ commands that can query and manage the DHCP Server service on LON-DC1.

You have to discover commands that can list existing DHCP leases. Because a DHCP server can host multiple address scopes, you will also have to find a command that can list the available scopes.

In production networks, it is common for new servers to be temporarily attached to a dedicated network subnet for provisioning purposes. These subnets are also known as *bench networks*, *lab networks*, and other names. Your lab environment emulates this kind of environment, and so when you discover what DHCP scope is available for your use, you will hard-code that DHCP scope ID into your script.

### Create a DHCP Reservation

Organizations differ in their preferred method of IP address management for servers. Some organizations assign static IP addresses to servers, and manually track those assignments. Other organizations leave servers set to use a dynamic IP address, and configure the DHCP server with a reservation that assigns the same IP address to a server every time that the server requests an address. Your lab will use a reservation.

You will therefore have to discover a command that enables you to add a DHCP reservation to the DHCP server that is running on LON-DC1. That command will likely need the DHCP computer name, the DHCP scope ID in which to create the reservation, the IP address to assign, and the MAC address (also known as the *client ID*) that the IP address will be issued to.

As you work, remember that creating the reservation will not automatically cause the server to begin using it. The server will continue to use its current IP address until that lease is released or must be renewed, or until the server is restarted. Your script will therefore have to use the appropriate IP address at the appropriate time when it tries to communicate with the new server.

## Modify the TrustedHosts List

By default Windows Server 2012 computers are configured to allow for relatively few communications into the server. Technologies such as Windows Management Instrumentation (WMI) may not be usable by default because of Windows Firewall restrictions. The one administrative communication technology that is enabled by default is Windows Remote Management (WinRM), which uses the Web Services for Management, or WS-Management (WS-MAN), protocol.

A problem with WinRM is that it typically demands mutual authentication between you and the server that you are connecting to. In a domain environment, this is provided natively by the Active Directory® Kerberos authentication protocol. Your new Server Core computer, however, has not yet been joined to the domain. One alternative would be to install a Secure Sockets Layer (SSL) encryption certificate on the computer, because such a certificate can also provide mutual authentication. However, that installation is difficult to perform remotely.

A second alternative is to temporarily modify the WinRM TrustedHosts list on the client computer so that it no longer demands mutual authentication. This would enable you to connect to the Server Core computer to begin provisioning it. You will want to save any existing TrustedHost list values before modifying the list, and restore the original list after you have completed the provisioning process.

The TrustedHosts list on your computer can be accessed by using the PSDrive **WSMAN**. In that drive, you would navigate to **WSMAN:\localhost\Client** to locate the TrustedHosts item. Typically, the **Get-Content** and **Set-Content** commands can be used to modify the contents of the item. Because this item is security-sensitive, Windows PowerShell prompts you before changing it.

## Add the Server to the Domain

When you can communicate with the new computer, you will add it to the domain. You can do this with a single Windows PowerShell command, although you will be unable to run that command on the client computer because the Server Core computer is not yet configured to receive that communication. Instead, you will use Windows PowerShell remoting to send the command over the network to the new Server Core computer, and the Server Core computer will execute the command locally. Part of the command's feature set includes the ability to automatically restart the computer, which finishes the process of joining it to the domain.

Sending commands to remote computers can be especially difficult when you also have to send variable information, such as the computer's new name, or a domain credential, both of which you will have to send. Remember that variables defined on the client computer have no meaning on the remote computer, and cannot be included in the **-ScriptBlock** parameter of **Invoke-Command**. However, the **Invoke-Command** cmdlet provides a specific way to pass variables' contents from the client computer to the remote computer. That technique is implemented by the command's **-ArgumentList** parameter, together with a **Param()** block inside the **-ScriptBlock** parameter's value.

After you send the command to the Server Core computer, you will have to disconnect from it (in anticipation of its restart) and then wait for some time for the restart to be completed. One way of robustly waiting for the restart to complete is to try to continually reconnect to the remote computer through Windows PowerShell remoting. A ping is unreliable because the remote computer's Windows Firewall does not allow the Internet Control Message Protocol (ICMP) traffic by default. However, to simplify this lab, you will tell your script to wait for five minutes after telling the remote computer to restart. That should give the remote computer time to restart and start accepting connections again.

## Add a Role to the Server

When the computer is in the domain, you will be able to connect to it more easily by using your regular Domain Administrator credential. You will have to connect one more time in order to tell the computer to add a role to itself. For the purposes of this lab, you will add only a single role, although in a production environment, you might want to add multiple roles.

### Restore TrustedHosts List

Finally, you will restore the client computer's TrustedHosts list to its original value (even if that value was blank). This means that the IP address originally used by the new Server Core computer is no longer trusted without mutual authentication, reducing the possibility that a sensitive user credential will be accidentally delegated to any computer at that IP address in the future.

### Lab Notes

If you have difficulties or run out of time on an exercise, you can move on. Your Lab Answer Key includes instructions for copying a starting point from the lab files to your Modules folder. Be sure to follow the instructions carefully.

- Starting points are provided for each exercise in this lab.
- If you have difficulties and want to move on, carefully follow the instructions in your Lab Answer Key to copy the starting point into your Modules folder.

## Lab: Provisioning a New Server Core Instance

### Scenario

You are an administrator who is preparing a new Windows Server 2012 Server Core instance for use within the network. Windows Server 2012 was installed in a new virtual machine, and a local Administrator password was set. All other configuration settings on the new computer were left at their default settings. You must use Windows PowerShell to perform several basic provisioning tasks.

### Objectives

After completing this lab, students will be able to:

- Discover a new virtual machine's dynamic IP address by querying a DHCP server
- Create a DHCP reservation for a virtual machine
- Rename a new virtual machine and join it to the domain
- Add the Web Server role to the a virtual machine

### Lab Setup

Estimated Time: 120 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1,

- User name: Administrator
- Password: Pa\$\$w0rd
- Domain: ADATUM

Virtual Machine: 10961B-LON-SVR1

- User Name: LON-SVR1\Administrator
- Password: Pa\$\$w0rd

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

1. On the host computer, hover the mouse pointer over the bottom left corner of the taskbar, click on the resultant **Start menu**, and then click **Hyper-V Manager** on the **Start Screen**.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 and 3 for 10961B-LON-CL1.
6. Repeat steps 2 and 3 for 10961B-LON-SVR1, and sign in with the local user name Administrator and password Pa\$\$w0rd.

## Exercise 1: Create a Parameterized Script

### Scenario

In this exercise, you will create the beginnings of a parameterized script. As you progress through the other exercises in this lab, you will add to this script, gradually building more and more functionality until the script is completed.

The main tasks for this exercise are as follows:

1. In the Script Pane, create a script named Set-ServerCoreInstance.ps1 that uses cmdlet binding and has a parameter block
2. Add input parameters to the script. Perform this task in the Script Pane

► **Task 1: In the Script Pane, create a script named Set-ServerCoreInstance.ps1 that uses cmdlet binding and has a parameter block**

1. Ensure you are signed in to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Open the Windows PowerShell Integrated Scripting Environment (ISE) as Administrator.
3. Ensure that the execution policy is set to **RemoteSigned**.
4. Create a new script in the scripting pane in the ISE.
5. Add the cmdlet binding attribute to the script, and add a parameter block to the script.
6. Save the script as **C:\Scripts\Set-ServerCoreInstance.ps1**.

► **Task 2: Add input parameters to the script. Perform this task in the Script Pane**

1. In your Set-ServerCoreInstance.ps1 script, add input parameters for the following:
  - **MACAddress**, a single mandatory string.
  - **LocalCredential**, having a default value that is a parenthetical expression that prompts for a credential object, ensuring that the prompt includes the message "Local Credential."
  - **DomainCredential**, having a default value that is a parenthetical expression that prompts for a credential object, ensuring that the prompt includes the message "Domain Credential."
  - **NewComputerName**, a single mandatory string.
  - **NewIPAddress**, which is a single mandatory string.
  - **Role**, a single mandatory string.
  - **Domain**, a single string that defaults to "**ADATUM**".
  - **ScopeID**, a single mandatory string that has no default value.
  - **DHCPServerName**, a single mandatory string that has no default value.

**Results:** After completing this exercise, you will have created the beginnings of a parameterized script.

## Exercise 2: Get the Dynamic IP Address of the New Server Core Computer

### Scenario

In this exercise, you will use the MAC address of the new Server Core virtual machine to retrieve its IP address from the DHCP server. You will first perform this task interactively in the shell, and then add a command to your script to perform this same task.

The main tasks for this exercise are as follows:

1. Discover the MAC address of the LON-SVR1 computer
2. In the Console pane, look up the IP address of the Server Core instance in the DHCP server
3. Update your script to look up the IP address

#### ► Task 1: Discover the MAC address of the LON-SVR1 computer

**Note:** In a real environment, you would have access to the MAC address either on a label (for a physical computer) or through hypervisor configuration tools (for a virtual machine). In this task, you will discover the information manually.

1. Log on to LON-SVR1 as **Administrator** by using the password **Pa\$\$w0rd**.
2. Run **Ipconfig /all** and note the 12-character physical address. Make sure that you include the dashes between each of the 6 address elements. *You will need this physical (MAC) address information throughout this lab, so make sure you write it down clearly, in an easy-to-access location.*
3. Log off from the virtual machine.
4. In the Console pane on LON-CL1, save the MAC address in the variable **\$MACAddress**.

**Note:** You will not log on to LON-SVR1 again.

#### ► Task 2: In the Console pane, look up the IP address of the Server Core instance in the DHCP server

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. In the Windows PowerShell ISE, display the Console pane.
3. Using a keyword such as **Lease**, discover the command that will query a DHCP lease from the DHCP server. Read the Help for the command, and notice that it requires you to specify a DHCP scope.
4. Using a keyword such as **Scope**, discover the command that will list DHCP scopes from the server LON-DC1.
5. Place the string value **LON-DC1** into a variable named **\$DHCPServerName**.
6. Display a list of DHCP scopes on the server LON-DC1. Write down the IP address of the only scope.
7. Save the scope IP address in the variable **\$ScopeID**.
8. Display a list of all leases in the only DHCP scope. Use the variables **\$DHCPServerName** and **\$ScopeID** in your command.
9. Display the IP address of the Server Core instance. The IP address must be displayed as a simple string, not as a complex, multiple property object. Save the IP address in a string variable named **\$OldIPAddress**.

You will have to run 3–4 commands in a pipeline to perform this task.

- You will need to use the **\$MACAddress** and **\$OldIPAddress** variables that you created in previous tasks. You will also need to use the **\$ScopeID** and **\$DHCPServerName** variables that you created in this task.
  - Use **Where-Object** to filter the list of leases so that only the lease having the same client ID (MAC address) as the Server Core instance is produced as output.
  - Remember that **Select-Object** has a parameter, **-ExpandProperty**, that can extract the contents of a single property from a complex, multiple property object.
  - You will have to use **Select-Object** twice. The first time, expand the IP address. Pipe that to **Select-Object** again to expand the **IPAddressToString** property.
10. Place the **\$OldIPAddress** variable inside double quotation marks, and assign the resulting string to **\$OldIPAddress**. This will convert the IP address to a string object.
  11. Create a variable named **\$NewComputerName** that contains the value **LON-SVR2**.

**Note:** The existing name of the target computer is LON-SVR1. We will rename it to the name defined in the variable **\$NewComputername** as part of this process i.e. LON-SVR2.

► **Task 3: Update your script to look up the IP address**

1. Update your script to look up the IP address.
2. Add the commands from steps 8 to 10 of the previous task to your script.

**Results:** After completing this exercise, you will have written a command to retrieve an IP address from the DHCP server based on the MAC address of the computer.

## Exercise 3: Create a DHCP Reservation for the Server Core Instance

### Scenario

In this exercise, you will create a DHCP reservation in the DHCP server for the Server Core instance's MAC address. You will set it to use the IP address 10.0.0.10. This new address will not take effect immediately, but will instead take effect the next time that the virtual machine is restarted.

The main tasks for this exercise are as follows:

1. In the Console pane, create the DHCP reservation
2. In the Script Pane, update your script to create the DHCP reservation

► **Task 1: In the Console pane, create the DHCP reservation**

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Using a keyword like **Reservation**, find the command that will create a new reservation.
3. In the variable **\$NewIPAddress**, store the string value **10.0.0.10**.
4. Run a command that will create a DHCP reservation for the Server Core instance's physical (MAC) address. Use the variables **\$MACAddress**, **\$ScopeID**, **\$DHCPServerName**, and **\$NewIPAddress** in your command.

**Note:** You can create the DHCP reservation only once. If your command succeeds, move on to the next task.

► **Task 2: In the Script Pane, update your script to create the DHCP reservation**

1. Add the command from step 3 of the previous task to the end of your script.
2. Save your script.

**Results:** After completing this exercise, you will have created a DHCP reservation for the LON-SVR1 computer.

## Exercise 4: Modify the local TrustedHosts list

### Scenario

In this exercise, you will read your current TrustedHosts list into a variable for later use. You will then add the IP address of the Server Core computer to the TrustedHosts list, bypassing the need for mutual authentication when you connect to that IP address by using Windows PowerShell Remoting.

The main tasks for this exercise are as follows:

1. Save the TrustedHosts list to a variable. Perform this task in the Console pane
2. In the Console pane, add the Server Core computer's IP address to the TrustedHosts list
3. In the Script Pane, update your script to modify TrustedHosts

► **Task 1: Save the TrustedHosts list to a variable. Perform this task in the Console pane**

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Save the value of WSMAN:\Localhost\Client\TrustedHosts to the variable **\$OriginalTrustedHosts**. Make sure that you save *only* the value, not the whole item object.
3. Verify the contents of **\$OriginalTrustedHosts**.

► **Task 2: In the Console pane, add the Server Core computer's IP address to the TrustedHosts list**

1. Run the **Set-Item** command to add the IP address in **\$OldIPAddress** to your TrustedHosts list. Notice that the TrustedHosts list requires string values. You must verify the kind of object in **\$OldIPAddress** and convert it if necessary.
2. Verify that the TrustedHosts list has changed.
3. So you can test your ability to restore the list, reset the TrustedHosts list back to the value stored in **\$OriginalTrustedHosts** and verify the change.
4. So that you can proceed to the next step, set TrustedHosts to **\$OldIPAddress** and verify it was set correctly.

► **Task 3: In the Script Pane, update your script to modify TrustedHosts**

1. Modify your script to include cmdlet binding attribute with a parameter block.
2. Save the TrustedHosts list to a variable.
3. Add several blank lines to the end of your script.
4. At the end of your script, add the command from step 1 of the previous task. This command must remain the last command in your script. Future additions to your script must be added before this command.

**Results:** After completing this exercise, you will have saved your TrustedHosts list, and added the Server Core computer's IP address to it.

## Exercise 5: Add a Role to the Server Core Instance

### Scenario

In this exercise, you will run a Windows PowerShell command that adds a designated role to the Server Core computer.

The main tasks for this exercise are as follows:

1. In the Console pane, add a role to the Server Core computer
2. In the Script Pane, update your script to add a role

► **Task 1: In the Console pane, add a role to the Server Core computer**

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Using a keyword like **feature**, find a command that can add a Windows Feature (role) to a computer.
3. Write a command that runs **Get-Credential** to prompt for a credential, and then stores it in **\$LocalCredential**. When prompted, provide the user name **Administrator** and the password **Pa\$\$w0rd**.
4. Write a command that uses **Invoke-Command** to add the **Telnet-Client** role to the Server Core instance. Refer to the Server Core instance by using the IP address in **\$OldIPAddress**. Provide the credential in **\$LocalCredential**.

► **Task 2: In the Script Pane, update your script to add a role**

1. Modify your script to include the commands from step 3 of the previous task.
2. These commands must be added before the command that restores the TrustedHosts list.

**Results:** After completing this exercise, you will have added a role to the Server Core computer.

## Exercise 6: Add the Server Core Instance to the Domain

### Scenario

In this exercise, you will use Windows PowerShell Remoting to transmit a command to the Server Core computer. That command, when it is executed locally by the Server Core computer, will rename the computer, join it to the ADATUM domain, and restart it.

The main tasks for this exercise are as follows:

1. In the Console pane, add the computer to the domain
2. In the Script Pane, update your script to add and rename the computer

► **Task 1: In the Console pane, add the computer to the domain**

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.

2. Using a keyword like **computer**, find a command that will let you add a computer to a domain.
3. Write a command that runs **Get-Credential** to prompt for a credential, and stores that credential in **\$DomainCredential**. When prompted, provide the user name **ADATUM\Administrator** and the password **Pa\$\$w0rd**.
4. Write a command that uses **Invoke-Command** to tell the Server Core instance to add itself to the domain, to rename itself, and to restart itself. You will need to provide the following information to the command:
  - o **Invoke-Command** will need four parameters:
    - The IP address in **\$OldIPAddress**
    - The credential in **\$LocalCredential**
    - A script block that includes a parameter block containing two parameters
    - An argument list that includes two parameters, **\$NewComputerName** and **\$DomainCredential**
  - o The command inside the script block of **Invoke-Command** will require the following:
    - The domain name that the computer will be added to
    - The computer's new name
    - A credential for the domain
    - Instruction to restart

► **Task 2: In the Script Pane, update your script to add and rename the computer**

1. Add the command from step 3 of the previous task to your script. The command must be added before the command that restores the TrustedHosts list.
2. Save your script.

**Results:** After completing this exercise, you will have renamed the Server Core computer and added it to the domain.

## Exercise 7: Test the Completed Script

### Scenario

In this final exercise, you will run your complete script and verify its results.

The main tasks for this exercise are as follows:

1. Run the completed script in the Console pane
2. To prepare for the next module

► **Task 1: Run the completed script in the Console pane**

1. Revert the **10961B-LON-DC1** and the **10961B-LON-SVR1** virtual machines. You may also need to restart 10961B-LON-DC1 and 10961B-LON-SVR1, and then reset the execution policy to remote signed on 10961B-LON-CL1 after reverting them.
2. In the Console pane, run your C:\Scripts\Set-ServerCoreInstance.ps1 script in a way that prevents Windows PowerShell from prompting you for anything except the two credentials and any confirmation prompts.

**Note:** It is expected that the target computer will restart and you will lose your connection. In the console prompt you will see an error message saying “..Connection Lost..” attempting to reconnect.

3. Use **Get-ADComputer** to verify that the new computer is in the domain.
4. Use **Get-WindowsFeature** to verify that the Telnet-Client role is installed on LON-SVR1.
5. Use **Dir** to verify that the local TrustedHosts setting is empty.

► **Task 2: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have automated the provisioning process and verified the results.

**Question:** Why was it necessary to modify TrustedHosts?

**Question:** Why was it necessary to use the **param()** block and **-ArgumentList** with **Invoke-Command**?

## Module Review and Takeaways



**Best Practice:** Always try to test commands in the console before adding them to a script. In this manner, you have to write and debug only one command at a time. This makes it easier to assemble a working, bug-free script.

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
One computer cannot connect to another.	
An extracted property value is not a string.	

# Module 11

## Using Background Jobs and Scheduled Jobs

### Contents:

Module Overview	11-1
<b>Lesson 1: Using Background Jobs</b>	<b>11-2</b>
<b>Lab A: Using Background Jobs</b>	<b>11-8</b>
<b>Lesson 2: Using Scheduled Jobs</b>	<b>11-11</b>
<b>Lab B: Using Scheduled Jobs</b>	<b>11-16</b>
Module Review and Takeaways	11-19

## Module Overview

In this module, you will learn about the job features of Windows PowerShell™. Jobs are an extension point in Windows PowerShell, meaning there are many different kinds of jobs. Each kind of job can work slightly differently, and each kind has different capabilities.



**Additional Reading:** You can read more about jobs at the TechNet about\_Jobs page, <http://go.microsoft.com/fwlink/?LinkId=306157>.

### Objectives

After completing this module, students will be able to:

- Create and manage background jobs
- Create and manage scheduled jobs

## Lesson 1

# Using Background Jobs

In this lesson, you will learn about three types of jobs. These three jobs form the basis of the Windows PowerShell job system.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose and use of background jobs
- Start jobs
- Manage jobs
- Retrieve job results

### What Are Background Jobs?

*Background jobs* are Windows PowerShell commands that run in the background. Each job's command results are stored in memory until you retrieve them.

There are three basic types of background job:

- *Local jobs* run their commands on the local computer. These jobs typically access only local resources. However, you can create a local job whose commands target a remote computer. For example, you could create a local job that includes the following command, whose **-ComputerName** parameter makes it connect to a remote computer:

```
Get-Service -Name * -ComputerName LON-DC1
```

- Run commands in the background
- Store command results in memory for retrieval
- Three basic job types:
  - Local
  - Remoting
  - WMI
- Each job type has different characteristics

- *Remoting jobs* use Windows PowerShell remoting to transmit their commands to one or more remote computers. The commands are run on those remote computers, and the results are returned to the local computer and stored in memory. This kind of job requires that Windows PowerShell remoting be enabled on the remote computers, although in Windows PowerShell version 3.0 and later, you do not need to enable remoting on the computer where the job is created. Windows PowerShell Help files refer to this kind of job as a *Remote job*.
- *WMI jobs* use Windows Management Instrumentation (WMI). The command runs on your computer but may connect to one or more remote computers' WMI services.

Each type of job has different characteristics. For example, local and remoting jobs run in a background Windows PowerShell runspace. You can think of them as running in a hidden instance of Windows PowerShell. Other types of jobs may have different characteristics. Also, add-in modules can add more job types to Windows PowerShell, and those job types will have their own characteristics.

Remoting jobs are the most useful for managing multiple remote computers. Because remoting transmits commands to remote computers, and because the remote computers run those commands by using their own local resources, nearly any command can be included in the job.

Remember that these are not the only kinds of jobs that Windows PowerShell can contain. Modules and other add-ins can create additional job types.

## Starting Jobs

You start each of the three basic job types in a different way.

### Local Jobs

Start local jobs by running **Start-Job**. Provide either the **-ScriptBlock** parameter to specify a single command line or a small number of commands. Provide the **-FilePath** parameter to run an entire script on a background thread.

By default, jobs are given a sequential job identification (ID) number and a default job name. You cannot change the job ID number assigned, but you can use the **-Name** parameter to specify a custom job name. Custom names make it easier to retrieve the job and easier to identify the job in the job list.



**Note:** At first, job ID numbers may not seem to be sequential. You will learn why later in this module.

- Local jobs:

```
Start-Job -ScriptBlock { Dir }
```

- Remoting jobs:

```
Invoke-Command -ScriptBlock { Get-Service }  
-ComputerName LON-DC1 -AsJob
```

- WMI jobs:

```
Get-WmiObject -Class Win32_BIOS  
-ComputerName LON-DC1 -AsJob
```

You can specify the **-Credential** parameter to have the job run 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 so on.

Some examples:

```
PS C:\> Start-Job -ScriptBlock { Dir C:\ -Recurse } -Name LocalDirectory  
Id     Name          PSJobTypeName   State    HasMoreData  Location  
--     --          -----          ----      -----      -----  
2      LocalDirectory BackgroundJob  Running   True        localhost  
PS C:\> Start-Job -FilePath C:\test.ps1 -Name TestScript  
Id     Name          PSJobTypeName   State    HasMoreData  Location  
--     --          -----          ----      -----      -----  
4      TestScript    BackgroundJob  Running   True        localhost
```

### Remoting Jobs

Start remoting jobs by running **Invoke-Command**. This is the same command you would use to send commands to a remote computer, and you learned about this command in Module 9, "Administering Remote Computers." Add the **-AsJob** parameter to make the command run in the background, and use the **-JobName** parameter to specify a custom job name. All other parameters of **Invoke-Command** are used in the same way. For example:

```
PS C:\> Invoke-Command -ScriptBlock { Get-EventLog -LogName System -Newest 10 }  
-ComputerName LON-DC1,LON-CL1,LON-SVR1 -AsJob -JobName RemoteLogs  
Id     Name          PSJobTypeName   State    HasMoreData  Location
```

--	---	-----	-----	-----	-----
6	RemoteLogs	RemoteJob	Running	True	LON-DC1...

 **Note:** Notice that the **-ComputerName** parameter is a parameter of **Invoke-Command**, not of **Get-EventLog**. The parameter causes the local computer to coordinate the remoting connections to the three computers specified. Each computer receives only the **Get-EventLog** command and runs it locally, returning the results.

Remoting jobs are created and managed by the computer where **Invoke-Command** is run. You can refer to that computer as the *initiating computer*. The commands inside the job are transmitted to remote computers. The remote computers execute the job and return the results to the initiating computer. The initiating computer stores the results of the job in its memory.

## WMI Jobs

Start a WMI job by running **Get-WmiObject**. This is the same command you would use to query WMI instances from a remote computer, and you learned about this command in Module 6, "Querying Management Information by Using WMI and CIM." Add the **-AsJob** parameter to have the command run on a background thread. There is no way to provide a custom job name. The Help file for **Get-WmiObject** states the following for the **-AsJob** parameter:

Note: To use this parameter with remote computers, the local and remote computers must be configured for remoting. Additionally, you must start Windows PowerShell by using the "Run as administrator" option in Windows Vista and later versions of Windows. For more information, see about\_Remote\_Requirements.

However, the Help file is not entirely correct about this behavior. WMI jobs do not require remoting to be enabled on either the initiating computer or on the remote computer. WMI jobs do require that WMI be accessible on the remote computers.

Here is an example:

```
PS C:\> Get-WmiObject -Class Win32_NTEventLogFile -ComputerName localhost,LON-DC1 -AsJob
Id      Name          PSJobTypeName   State       HasMoreData    Location
--      --           -----          -----        -----         -----
10     Job10        WmiJob        Running      True          Localho...
```

## Job Objects

Notice that each of the preceding examples produces a job object as their result. The *job object* represents the running job, and can be used to monitor and manage the job.

## Managing Jobs

When you start a job, you are given a job object. That object can be used 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, no matter how many computers the job connects to. The parent job contains one or more *child jobs*. Each child job represents a single computer. In a local job, there will be only one child job. Remoting jobs and WMI jobs will have one child job for each computer that you specified.

- **Get-Job**
  - Add **-ID** to retrieve specific job by ID
  - Add **-Name** to retrieve specific job by name

- To get a list of child jobs:
 

```
Get-Job -ID <parent_ID> | Select -ExpandProperty ChildJobs
```

- **Stop-Job**
- **Remove-Job**
- **Wait-Job**

### Retrieving Jobs

You can list all current jobs by running **Get-Job**. You can list a specified job by adding the **-ID** or **-Name** parameter and specifying the desired job ID or job name. Using the job ID, you can also retrieve child jobs. For example:

```
PS C:\> Get-Job
Id Name PSJobTypeName State HasMoreData Location
-- ---- ----- ---- -----
2 LocalDirectory BackgroundJob Running True localhost
4 TestScript BackgroundJob Completed True localhost
6 RemoteLogs RemoteJob Failed True LON-DC1...
10 Job10 WmiJob Failed False localhost...
PS C:\> Get-Job -Name TestScript
Id Name PSJobTypeName State HasMoreData Location
-- ---- ----- ---- -----
4 TestScript BackgroundJob Completed True localhost
PS C:\> Get-Job -ID 5
Id Name PSJobTypeName State HasMoreData Location
-- ---- ----- ---- -----
5 Job5 BackgroundJob Completed True localhost
```

Notice that each job has a status. Parent jobs always display the status of the worst child job. In other words, if a parent contains 4 child jobs, and 3 of those jobs finished successfully but 1 of those jobs failed, the parent job status will be **Failed**.

### Listing Child Jobs

You can list the child jobs of a specified parent job by retrieving the parent job object and expanding its **ChildJobs** property. For example:

```
PS C:\> Get-Job -Name RemoteLogs | Select -ExpandProperty ChildJobs
Id Name PSJobTypeName State HasMoreData Location
-- ---- ----- ---- -----
7 Job7 RemoteJob Failed False LON-DC1
8 Job8 RemoteJob Completed True LON-CL1
9 Job9 RemoteJob 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 later. In Windows PowerShell 3.0 and later, you can also use the **-IncludeChildJobs** parameter of **Get-Job** to display child jobs of a job.

## Managing Jobs

Windows PowerShell includes several commands used to manage jobs. Each of these can have one or more jobs piped to it, or you can specify jobs by using the **-ID** or **-Name** parameters. Both of those parameters accept multiple values, meaning you can specify a comma-separated list of job ID numbers or names. The job management commands include:

- **Stop-Job** stops a job that is running. Use this command to cancel a job that is in an infinite loop or that has run longer than you want.
- **Remove-Job** deletes a job object, including any command results that were stored in memory. You should use this when you are finished working with a job so that the shell can free up memory.
- **Wait-Job** is typically used in a script. It pauses script execution until the specified jobs reach the specified status. You can use this in a script to start several jobs, and then make the script wait until those jobs complete before continuing.

Remoting, WMI, and local jobs are managed inside the Windows PowerShell process. When that process ends, all jobs and their results are removed and can no longer be accessed.

## Retrieving Job Results

As a job runs, any output produced by its commands is stored in memory. Output is stored beginning with the first output produced by the command. You do not have to wait for a command to complete before output may be available.

The job list shows whether a job has stored results waiting. For example:

```
PS C:\> Get-Job
Id      Name          PSJobTypeName
State   HasMoreData   Location
--      ----
-----  -----
13     Job13        BackgroundJob  Running    True      localhost
```

- Use **Receive-Job**
  - Pipe jobs to it to specify jobs
  - Use **-ID** to specify by job ID
  - Use **-Name** to specify by job name

- Add **-Keep** to retain a copy of the results in memory. Otherwise, results are not retained in memory.
- Receiving results from a parent job will receive results from all child jobs.

In this example, job ID 13 is still running, but the **HasMoreData** column indicates that results have already been stored in memory.

To receive the results of a job, use the **Receive-Job** command.

By default, job results are not stored in memory after they are delivered to you. That means you can use **Receive-Job** only once per command. Add the **-Keep** parameter to retain a copy of the job results in memory so that you can retrieve them again.

If you retrieve the results of a parent job, you will receive the results from all of its child jobs. You can also retrieve the results of a single child job, or of multiple child jobs, if needed.

You can retrieve the results of a job that is still running. However, unless you specify **-Keep**, the job object's results will be empty until the job's command adds new output.

For example:

```
Receive-Job -ID 13 | Format-Table -Property Name,Length
```

## Demonstration: Using Background Jobs

In this demonstration, you will see how to create and manage two kinds of jobs.

### Demonstration Steps

1. Enable remoting on LON-CL1.
2. Start a local job that produces a complete directory listing for **C:\**, including subfolders. Give the job the name **LocalDir**.
3. Start a remoting job that queries the most recent 100 entries from the Security event log on LON-CL1 and on LON-DC1. Give the job the name **RemoteLogs**.
4. Display a list of running jobs.
5. Stop the **LocalDir** job.
6. Retrieve the results from the **LocalDir** job.
7. Remove the **LocalDir** job.
8. Display a list of running jobs, and repeat this step every few minutes until the **RemoteLogs** job completes.
9. Display a list of child jobs under the **RemoteLogs** job.
10. Retrieve the results from LON-DC1, keeping a copy of the results in memory.
11. Retrieve the results from the **RemoteLogs** parent job.
12. Remove the **RemoteLogs** job.

**Question:** What are some tasks that you might want to run in the background?

# Lab A: Using Background Jobs

## Scenario

You need to perform several tasks that can run concurrently. You will start these tasks as background jobs.

## Objectives

After completing this lab, students will be able to:

- Start jobs
- Manage jobs

## Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click 10961B-LON-DC1, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. Perform the Labs steps on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Starting Jobs

### Scenario

In this exercise, you will start jobs using two of the basic job types.

The main tasks for this exercise are as follows:

1. Start a Remoting Job
2. Start a Local Job

### ► Task 1: Start a Remoting Job

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.

2. Enable remoting on LON-CL1, if it is not already enabled.
3. Start a remoting job that retrieves a list of physical network adapters from LON-DC1 and LON-CL1. Name the job **RemoteNetAdapt**.
4. Start a remoting job that retrieves a list of Server Message Block (SMB) shares from LON-DC1 and LON-CL1. Name the job **RemoteShares**.
5. Start a remoting job that retrieves all instances of the **Win32\_Volume** CIM class from every computer in Active Directory® Domain Services. Name the job **RemoteDisks**. Because some domain computers may not be started, some child jobs may fail.

► **Task 2: Start a Local Job**

1. Start a local job that retrieves all entries from the Security event log. Name the job **LocalSecurity**.
2. Using the range operator (..) and **ForEach-Object**, start a local job that produces 100 directory listings of drive C, including subfolders. Name the job **LocalDir**. Proceed to the next task while this job is still running.

**Results:** After completing this exercise, you will have started jobs using two of the basic job types.

## Exercise 2: Managing Jobs

### Scenario

In this exercise, you will manage the jobs that you created in the previous exercise.

The main tasks for this exercise are as follows:

1. Review Job Status
2. Stop a Job
3. Retrieve Job Results

► **Task 1: Review Job Status**

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. Display a list of running jobs.
3. Display a list of running jobs whose names start with **remote**.

► **Task 2: Stop a Job**

- Force the **LocalDir** job to stop.

► **Task 3: Retrieve Job Results**

1. Wait for all remaining jobs to either complete or fail.
2. Receive the results of the **RemoteNetAdapt** job.
3. Use a single command-line to receive the results from LON-DC1 for the **RemoteDisks** job.

You will need to start by querying the parent job, and then expanding its **ChildJobs** property. Filter the child jobs so that just the LON-DC1 job remains, and then receive the results from that job. You will use a total of four commands to complete this step.

MCT USE ONLY. STUDENT USE PROHIBITED

**Results:** After completing this exercise, you will have managed the jobs that you created in the previous exercise.

**Question:** `Get-CIMInstance` does not have an `-AsJob` parameter. Why? How would you use it in a job?

**Question:** What are some potential performance concerns about background jobs?

## Lesson 2

# Using Scheduled Jobs

In this lesson, you will learn to use scheduled jobs. Scheduled jobs are a new type of job that was introduced in Windows PowerShell 3.0.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose and use of scheduled jobs
- Create job options
- Create job triggers
- Create scheduled jobs
- Retrieve scheduled job results

### What Are Scheduled Jobs?

Scheduled jobs are a new kind of job that was introduced in Windows PowerShell 3.0.

Scheduled jobs consist of three components:

- The job itself defines the command that will run.
- Job options define options and execution criteria.
- Job triggers define when the job will run.

You typically create a job option object and a job trigger object, and store those objects in variables.

You then use those variables when creating the actual scheduled job.

• Require Windows PowerShell 3.0

• Three components:

- Job definition
- Job options
- Job triggers

 **Note:** Windows PowerShell includes two similar modules, **PSScheduledJob** (available whenever Windows PowerShell 3.0 is installed) and **ScheduledTasks** (available only on Windows Server® 2012 and Windows 8 and later). This module will focus on the commands in **PSScheduledJob**. You can learn more about this topic by running **help about\_scheduled\*** in the shell.

Scheduled jobs are a combination of Windows PowerShell background jobs and Windows® Task Scheduler tasks. Like background jobs, you define scheduled jobs in Windows PowerShell. Like tasks, job results are saved to disk, and tasks can run even if Windows PowerShell is not running.

The jobs you will create in this module can be managed by Windows PowerShell and seen in the Windows Task Scheduler. However, the commands you will learn in this module cannot manage other tasks that may appear in Windows Task Scheduler.



**Note:** The **ScheduledTasks** module includes commands that can manage all tasks in the Windows Task Scheduler. This module is included with Windows 8 and Windows Server 2012. You will not learn about this module in this course.

## 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. Some of those options include the following:

- **-HideInTaskScheduler** prevents the job from appearing in the Windows Task Scheduler. If you do not include this option, the final job will appear in the Windows Task Scheduler graphical user interface (GUI).
- **-RunElevated** configures the job to run under elevated permissions.
- **-WakeToRun** will wake the computer when the job is scheduled to run.

- Use **New-ScheduledJobOption** to create an option object
- Parameters correspond to options in the Task Scheduler GUI
- Options include:
  - **-RequireNetwork**
  - **-RunElevated**
  - **-WakeToRun**
  - **-HideInTaskScheduler**

Other parameters allow you to configure jobs that run when the computer is idle, and to configure other options. Many of the parameters correspond to options in the Windows Task Scheduler GUI.

To create a new option object and store it in a variable:

```
$opt = New-ScheduledJobOption -RequireNetwork -RunElevated -WakeToRun
```

You do not need to create an option object if you do not want to specify any of its configuration items.

## Job Triggers

A job trigger defines when a job will run. Each job can have multiple triggers. You create a trigger object by using the **New-JobTrigger** command. There are five basic types of trigger:

- **-Once** specifies a job that runs once. You can also specify a **-RandomDelay**, and you must specify the **-At** parameter to define when the job will run. That parameter accepts a **System.DateTime** object or a string that can be interpreted as a date.
- **-Weekly** specifies a job that runs weekly. You can specify a **-RandomDelay**, and you must specify both the **-At** and **-DaysOfWeek** parameters. **-At** accepts a date and time to define when the job will run. **-DaysOfWeek** accepts one or more days of the week to run the job. You will typically use **-At** to specify only a time, and then **-DaysOfWeek** to define the days for the job.
- **-Daily** specifies a job that runs every day. You must specify **-At** and provide a time when the job will run. You can also specify a **-RandomDelay**.

- Use **New-JobTrigger**
- Five basic types of trigger:
  - **-Once**
  - **-Weekly**
  - **-Daily**
  - **-AtLogOn**
  - **-AtStartup**

- **-AtLogOn** specifies a job that runs when the user logs on. This kind of job is similar to a logon script, except that it is defined locally rather than in the domain. You can specify **-User** to limit the user accounts that trigger the job, and **-RandomDelay** to add a random delay.
- **-AtStartUp** is similar to **-AtLogOn**, except that it runs the job when the computer starts. That typically runs the job before a user has an opportunity to log on.

For example, this command creates a trigger that runs on Mondays and Thursdays every week, at 3:00 P.M. 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. Or, you may specify **-FilePath** and provide the path and name of a Windows PowerShell script file that the job will run.
- **-Credential** is optional, and specifies the user account that will be used to run the job.
- **-InitializationScript** accepts an optional script block. The command or commands in that script block will execute before the job starts.
- **-MaxResultCount** is optional, and specifies the maximum number of result sets to store on disk. After this number is reached, the shell will delete older result sets to make room for new ones.
- **-ScheduledJobOption** accepts a job option object.
- **-Trigger** accepts a job trigger object.

- Use **Register-ScheduledJob**
- Creates job definition XML file on disk
- Parameters include:
  - **-Name**
  - **-ScriptBlock** or **-FilePath**
  - **-Credential**
  - **-MaxResultCount**
  - **-ScheduledJobOption** (job option object)
  - **-Trigger** (job trigger object)

To register a new job by using an option object in **\$opt** and a trigger object in **\$trigger**:

```
PS C:\> $opt = New-ScheduledJobOption -WakeToRun
PS C:\> $trigger = New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(5)
PS C:\> Register-ScheduledJob -Trigger $trigger -ScheduledJobOption $opt -ScriptBlock
{ Dir C:\ } -MaxResultCount 5 -Name "LocalDir"
Id      Name      JobTriggers      Command      Enabled
--      ----      -----      -----
1       LocalDir    1            Dir C:\        True
```

The resulting job is registered in the Windows Task Scheduler, and the job definition is created on disk. Job definitions are XML files stored in your profile folder, under `\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs`.

You can run **Get-ScheduledJob** to see a list of scheduled jobs on the local computer. If you know a scheduled job's name, you can use **Get-JobTrigger** and the **-Name** parameter to retrieve a list of that job's triggers.

## Retrieving Job Results

Because scheduled jobs can run when Windows PowerShell is not running, results are stored on disk in XML files. If the job was created by using the **-MaxResultCount** parameter, the shell will automatically delete old XML files to make room for new ones so that no more XML files exist than were specified for the **-MaxResultCount**.

After a scheduled job has run, running **Get-Job** in Windows PowerShell will show the results of the scheduled job as a job object.

- Run **Get-Job** to see a list of **PSScheduledJob** jobs
  - Each represents an execution of the scheduled job
  - Provides access to the job results
- Run **Receive-Job** to retrieve results
- Run **Remove-Job** to delete results and the job object

 **Note:** **Get-Job** will include scheduled jobs only if you first import the **PSScheduledJob** module by running **Import-Module PSScheduledJob**.

For example:

```
PS C:\> Get-Job
Id     Name      PSJobTypeName   State      HasMoreData    Location    Command
--     ---      -----          -----      -----        -----      -----
6      LocalDir  PSScheduledJob  Completed  True          localhost  Dir C:\
```

You can then use **Receive-Job** to receive the results of a scheduled job. If you do not specify **-Keep**, you can receive the results of a job only once per Windows PowerShell session. However, because the results are stored on disk, you can open a new Windows PowerShell session and receive the results again. For example:

```
PS C:\> Receive-Job -id 6 -Keep
Directory: C:\

Mode           LastWriteTime       Length Name
---           -----          -----
d---          7/26/2012 12:33 AM
d-r--         11/28/2012 1:54 PM
d-r--         12/28/2012 2:22 PM
d---          11/16/2012 9:33 AM
d---          9/18/2012 7:28 AM
d---          1/5/2013 7:49 AM
d---          1/5/2013 7:50 AM
d-r--         9/15/2012 8:16 AM
d---          12/19/2012 3:24 AM
-a---         1/1/2013 9:39 AM    2892628 EventReport.html
-a---         1/2/2013 12:37 PM      82 Get-DiskInfo.ps1
-a---         12/30/2012 12:33 PM      246 test.ps1
```

Each time the scheduled job runs, a new job object will be created to represent the results of the most recent job execution. You can use **Remove-Job** to remove a job. Doing so deletes the results file from disk. For example:

```
PS C:\> Get-Job -id 6 | Remove-Job
```

## Demonstration: Using Scheduled Jobs

In this demonstration, you will see how to create, run, and retrieve the results from a scheduled job.

### Demonstration Steps

1. Import the **PSScheduledJob** module.
2. Remove all jobs.
3. Create a job trigger that will run a job once in 2 minutes. Store the trigger in the variable **\$trigger**.
4. Using **\$trigger**, create a job named **DemoJob** that retrieves all entries from the Application event log on the local computer.
5. Display the triggers for the scheduled job. Notice the time that the job is scheduled to run.
6. Wait for the job to run. While waiting for the job to run, display a list of scheduled jobs.
7. After the job runs, display a list of jobs.
8. Receive the results of the **DemoJob** job.
9. Remove the **DemoJob** job results.

**Question:** Why might you use **Register-ScheduledJob** from the **PSScheduledJob** module instead of a command in the **ScheduledTasks** module?

## Lab B: Using Scheduled Jobs

### Scenario

You need to create a scheduled job that will perform a specified task on a daily basis.

### Objectives

After completing this lab, students will be able to:

- Create and manage a scheduled job

### Lab Setup

Estimated Time: 30 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click **10961B-LON-DC1**, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat step 2 for 10961B-LON-CL1.
6. Perform the Labs steps on the 10961B-LON-CL1 virtual machine.

### Exercise 1: Creating a Scheduled Job

#### Scenario

In this exercise, you will create and run a scheduled job, and retrieve the results from the job.

The main tasks for this exercise are as follows:

1. Create Job Options
2. Create a job trigger
3. Create a Scheduled Job
4. Retrieve Job Results
5. To prepare for the next module

► **Task 1: Create Job Options**

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. Create a job option object and store it in **\$option**. Configure the job object as follows:
  - o The job will wake the computer to run.
  - o The job will run under elevated permissions.

► **Task 2: Create a job trigger**

1. Create a job trigger object and store it in **\$trigger1**. Configure the trigger as follows:
  - o The job will run once in 10 minutes. Use **Get-Date** and a method of the resulting **DateTime** object to calculate 10 minutes from now.
2. Create a job trigger object and store it in **\$trigger2**. Configure the trigger as follows:
  - o The job will run at logon.

► **Task 3: Create a Scheduled Job**

1. Using **\$option** and **\$trigger1** create a new scheduled job having the following attributes:
  - o The job action retrieves all entries from the Security event log.
  - o The job name is **LocalSecurityLog**.
  - o The maximum number of job results is 5.
2. Using **\$option** and **\$trigger2** create a new scheduled job having the following attributes:
  - o The job action retrieves a list of running processes.
  - o The job name is **ProcList**.
3. Display a list of job triggers, including time, for the **LocalSecurityLog** scheduled job. Write down the time.
4. Log off from LON-CL1.
5. Wait until the time shown in step 3 has passed.
6. Log on to LON-CL1.

► **Task 4: Retrieve Job Results**

1. Open a Windows PowerShell console window.
2. Display a list of jobs.
3. Receive the results of the **LocalSecurityLog** job.
4. Receive the results of the **ProcList** job.

► **Task 5: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have created and run a scheduled job, and retrieved the results from the job.

**Question:** Is it possible to create a scheduled job without creating a job option object?

# Module Review and Takeaways

## Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
The <b>ScheduledTasks</b> module is not available.	

## Review Question(s)

**Question:** What is the main difference between a background job and a scheduled job?

## Real-world Issues and Scenarios

Remember that scheduled jobs are defined, stored, and managed locally. There is no means of central scheduled job management. For that reason, you should be careful to document job definitions. In large environments, you may prefer to use a centralized job management solution, such as Microsoft System Center Orchestrator.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 12

## Using Profiles and Advanced Windows PowerShell Techniques

### Contents:

Module Overview	12-1
<b>Lesson 1:</b> Using Advanced Windows PowerShell Techniques	12-2
<b>Lesson 2:</b> Creating Profile Scripts	12-11
<b>Lesson 3:</b> Working with Alternative Credentials	12-15
<b>Lab:</b> Practicing Advanced Techniques	12-18
Module Review and Takeaways	12-22

## Module Overview

In this module, you will learn about several advanced Windows PowerShell™ techniques and features. Many of these techniques and features extend functionality that you have learned about in previous modules. Some of these techniques are new and provide additional capabilities.

 You can read more about Windows PowerShell profile scripts at the TechNet [about\\_Profiles page](#):

<http://go.microsoft.com/fwlink/?LinkId=306158>

### Objectives

After completing this module, students will be able to:

- Manipulate data and objects by using advanced techniques and operators
- Create and manage profile scripts
- Connect to remote computers by using alternative credentials

## Lesson 1

# Using Advanced Windows PowerShell Techniques

This lesson will focus on string and date manipulation, advanced comparison operators, setting default parameter values, and running external commands. Although each of these topics is small by itself, they each provide valuable capabilities that you will use throughout your work with Windows PowerShell.

## Lesson Objectives

After completing this lesson, students will be able to:

- Manipulate string values
- Manipulate date values
- Convert date values in Windows Management Instrumentation (WMI) and Common Information Model (CIM) objects
- Compare objects by using advanced parameters
- Set default values for command parameters
- Execute external commands

## Manipulating String Values

In Windows PowerShell, string values are actually complete objects. They work the same as the other objects that you have used in this course. The type name for string objects is **System.String**; string objects include one property named **Length**. They also include many methods. Those methods provide many capabilities for string manipulation. Here are several examples:

```
PS C:\> $x = "Windows PowerShell"
PS C:\> $x.ToUpper()
WINDOWS POWERSHELL
PS C:\> $x.ToLower()
windows powershell
PS C:\> $x.Replace("o", "*")
Wind*ws P*werShell
PS C:\> $x.Length
18
PS C:\> $x.IndexOf("P")
8
PS C:\> $x.Substring(8, 5)
Power
```

- Strings are objects of the type **System.String**
- Run "`x` | Get-Member" to see the type's property and methods
- These methods are used for string manipulation
- The shell also offers three operators for string manipulation:
  - -Replace
  - -Split
  - -Join
- These all duplicate functionality of **System.String** methods

In these examples, you see how to use several of the methods of a **String** object.

 You can learn more about these and other methods by reading the documentation for the **System.String** class at this page:

<http://go.microsoft.com/fwlink/?LinkId=306159>

MCT USE ONLY STUDENT USE PROHIBITED



**Note:** Notice that the method name is always followed by parentheses, even when the method requires no arguments. There is no space between the method name and the parentheses. Properties, such as **Length**, do not use parentheses.

Windows PowerShell includes three operators that can be used for string manipulation:

- The **-Replace** operator searches for all occurrences of a substring and replaces them with another substring:

```
PS C:\> "Windows PowerShell" -replace "o", "***"
Wind***ws P***werShell
```

- The **-Join** operator accepts an array of strings, and concatenates them together. Each string is separated by a specified delimiter:

```
PS C:\> $items = "one", "two", "three", "four"
PS C:\> $items
one
two
three
four
PS C:\> $items -join "|"
one|two|three|four
```

- The **-Split** operator is the opposite of **-Join**. It accepts a delimited string and a delimiter, and returns an array of strings:

```
PS C:\> $line = "one,two,three,four"
PS C:\> $line
one,two,three,four
PS C:\> $line -split ","
one
two
three
four
```



**Note:** These Windows PowerShell operators provide functionality that overlaps the functionality of the string type's methods. You are free to use either operators or methods, depending upon what you are most comfortable with. Remember that the methods are typically case-sensitive, whereas these operators are case-insensitive.



**Note:** Some users will use **Get-Content** to read a comma-separated values (CSV) file. They will then use a **ForEach** loop to enumerate the lines of the file, and use **-Split** to break each line into individual columns. That approach is valid, but requires significant manual effort. The **Import-CSV** command returns more useful results with much less manual effort.

## Manipulating Date Values

Windows PowerShell recognizes date and time objects as a specific kind of object. The class name is **System.DateTime**, and the class offers several properties and methods that can be used to manipulate dates.

The **Get-Date** command produces an object of the type **System.DateTime**:

```
PS C:\> $today = Get-Date
PS C:\> $today
Friday, January 11, 2013 8:08:38 AM
PS C:\> $today.DayOfWeek
Friday
PS C:\> $today.Year
2013
PS C:\> $today.AddDays(-90)
Saturday, October 13, 2012 8:08:38 AM
PS C:\> $today.ToLocalTime()
Friday, January 11, 2013 8:08:38 AM
PS C:\> $today.ToShortDateString()
1/11/2013
PS C:\> $today.ToUniversalTime()
Friday, January 11, 2013 4:08:38 PM
```

- Dates are objects of the type **System.DateTime**
- Run **Get-Date** | **Get-Member** to see the type's properties and methods
  - The methods are used for date manipulation
  - The properties extract portions of a date or time
- Use the **[datetime]** type accelerator to convert string date representations to a **System.DateTime**
  - **[datetime]\$mydate = '1/1/2000'**

Windows PowerShell also offers a **[datetime]** type accelerator. If you assign this type to a variable, and then assign a string value to that variable, the shell will try to interpret the string as a date and create a **System.DateTime** object. The resulting object will have all the properties and methods of a **DateTime** object. For example:

```
PS C:\> [datetime]$mydate = '1/1/2000'
PS C:\> $mydate.DayOfWeek
Saturday
```

If you try to use this technique by using a string object that cannot be interpreted as a date or time, you will receive an error:

```
PS C:\> [datetime]$test = 'Hello'
Cannot convert value "Hello" to type "System.DateTime". Error: "The string was not
recognized as a valid DateTime.
There is an unknown word starting at index 0."
At line:1 char:1
+ [datetime]$test = 'Hello'
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

The properties and methods of **System.DateTime** are very useful for manipulating dates. For example, as shown here, you can calculate dates in the past and in the future by adding days, months, hours, milliseconds, seconds, and more.



**Note:** The **System.DateTime** class includes properties and methods for working with ticks. A *tick* represents 100 nanoseconds, or one ten-millionth of a second.



For more information about **DateTime Structure properties and methods**, read:

<http://go.microsoft.com/fwlink/?LinkID=306160>

## Working with WMI and CIM Dates

The objects that you retrieve by using **Get-CimInstance** will usually have date values in a **System.DateTime** format. For example, to query the local computer's last start time and display it as a short date, you would run this:

```
Get-CimInstance -ClassName  
Win32_OperatingSystem |  
Select  
@{n='LastStartDate';e={$PSItem.LastBoo  
tUpTime.ToShortDateString()}}
```

This example relies on the **LastBootUpTime** property of the **Win32\_OperatingSystem** class that contains a **System.DateTime**, and uses the **ToShortDateString()** method of that class.

However, objects retrieved by using **Get-WmiObject** expose dates and times in an internal WMI format. To make those easier to use, Windows PowerShell extends all WMI objects to include a **Convert.ToDateTime()** method. To duplicate this example by using **Get-WmiObject**, you would run this command:

```
Get-WmiObject -Class Win32_OperatingSystem |  
Select  
@{n='LastStartDate';e={$PSItem.Convert.ToDateTime($PSItem.LastBootUpTime).ToString("G")}}
```

This example uses the WMI object's **Convert.ToDateTime()** method. The object's own **LastBootUpTime** property is provided as an argument to the method. The resulting **System.DateTime** object's **ToString()** method is then used to retrieve the desired date representation.



**Note:** Remember that **\$PSItem** was introduced in Windows PowerShell 3.0. All versions of Windows PowerShell support **\$\_** instead of **\$PSItem**.

- Common Information Model (CIM) dates are in **System.DateTime** form
- Windows Management Instrumentation (WMI) dates are in an internal date format
  - All WMI objects have a **Convert.ToDateTime()** method that converts the internal format to a **System.DateTime**
  - They also have a **ConvertFromDateTime()** that converts a **System.DateTime** to WMI date format

## Advanced Operators

You have already learned about several Windows PowerShell comparison operators, such as **-eq**, **-ne**, **-gt**, and so on. In this lesson, you also learned about string manipulation operators like **-replace** and **-split**.

Windows PowerShell offers several other operators. Some of the most useful include the following:

- **-In** and **-Contains**. These operators test a collection to see whether the collection contains a particular object. A match is found only if the whole object specified is found in the collection specified. That means each of the object's property values must be the same in the collection for there to be a match. These operators are frequently used for simple objects such as strings. For example:

```
PS C:\> $collection = "one", "two", "three", "four"
PS C:\> $collection -contains "one"
True
PS C:\> $collection -contains "five"
False
PS C:\> "two" -in $collection
True
PS C:\> "six" -in $collection
False
```

- **-Match** is a regular expression operator. It returns **True** if the specified string matches the specified regular expression pattern. The operator also populates the built-in **\$Matches** array variable with the matches strings. Regular expression syntax is beyond the scope of this course, but here is an example:

```
PS C:\> "192.168.12.15" -match "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
True
PS C:\> "192.5522.12.15" -match "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
False
```

- **-As** tries to convert an object to a different object type. If the operator cannot perform the conversion, it may return an error or an empty value. For example:

```
PS C:\> 12345.67890 -as [int]
12346
PS C:\> "Microsoft" -as [int]
PS C:\>
```

Notice in the example for **-as** that the conversion of a fractional number to an integer produces a result that has been rounded to the nearest integer.

You can read more about these and other comparison operators by running **Help about\_comparison\_operators** in the shell. For more information about regular expressions, run **Help about\_regular\_expressions** in the shell.

- **-In** and **-Contains** test a collection to see if it contains an object
- **-Match** compares a string to a regular expression
- **-As** converts objects to a specified type

## Demonstration: Strings, Dates, and Operators

In this demonstration, you will see several advanced Windows PowerShell techniques.

### Demonstration Steps

1. Assign the string **Windows PowerShell** to the variable **\$x**.
2. Produce an uppercase version of **\$x**.
3. Produce a lowercase version of **\$x**.
4. Display the contents of **\$x**.
5. In **\$x**, replace every occurrence of **o** with **\*\*\***.
6. Display the number of characters in **\$x**.
7. Display the eighth through the thirteenth characters of **\$x**.
8. Use the **-replace** operator to replace every **o** in **\$x** with **-**.
9. Store today's date in the variable **\$today**.
10. Display the contents of **\$today**.
11. Display the day of the week of the date in **\$today**.
12. Display the date 30 days before the date in **\$today**.
13. Display a short date version of the date in **\$today**.
14. Using the **[datetime]** accelerator, store the date **1/1/1999** in the variable **\$mydate**.
15. Display the day of the week for the date in **\$mydate**.
16. Using a CIM command and the **Win32\_OperatingSystem** class, display the date that the local computer last started. Title the column **LastStartDate** and display the date as a short date string.
17. Using a WMI command and the **Win32\_OperatingSystem** class, display the date that the local computer last started. Title the column **LastStartDate** and display the date as a short date string.
18. Store an array of service names, as strings, in the variable **\$servicenames**.
19. Using the **-contains** operator, determine whether **\$servicenames** includes the value **WinRM**.
20. Using the **-in** operator, determine whether **\$servicenames** includes the value **MSSQLServer**.
21. Determine whether the string **192.168.12.15** matches the regular expression  
**\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}**.

## Setting Default Parameter Values

Windows PowerShell 3.0 introduced the ability to set persistent default values for the parameters of one or more commands. When you set a default value, you override any default value that the command would usually use internally. You can override your own default values by deleting them, changing them, or by manually providing parameter values when you run the command.

For example, suppose that you want to define a default value for the **-ComputerName** and **-ClassName** parameters of **Get-CimInstance**. You would create the first default value as follows:

```
$PSDefaultParameterValues = @{"Get-CimInstance:ComputerName"="LON-DC1"}
```

You are changing the built-in **\$PSDefaultParameterValues** variable. It consists of a hash table, and the hash table contains key-value pairs. Each pair is separated by a semicolon, although this example includes only one pair. The key consists of the command name, a colon, and the desired parameter name. The value is the default value that you want to assign. The syntax shown here completely replaces the existing contents of **\$PSDefaultParameterValues**.

To add a new parameter to the existing variable, use this syntax:

```
$PSDefaultParameterValues.Add("Get-CimInstance:ClassName","Win32_BIOS")
```

Again, you are providing a key/value pair to add to the hash table. However, the value and the key are specified as arguments to the **Add()** parameter. They are separated by a comma, not by an equal sign.

You can also display the contents of the variable:

```
PS C:\> $PSDefaultParameterValues
Name          Value
----          -----
Get-CimInstance:ComputerName  LON-DC1
Get-CimInstance:ClassName     Win32_BIOS
```

With the default settings, you can run the command and not provide those parameters:

```
PS C:\> Get-CimInstance
SMBIOSBIOSVersion : 6.00
Manufacturer      : Phoenix Technologies LTD
Name              : PhoenixBIOS 4.0 Release 6.0
SerialNumber      : 8f f8 9a 17 29 b2 98
Version           : INTEL - 6040000
PSComputerName    : LON-DC1
```

Notice that the computer name queried, and the class queried, are the ones specified in the default settings. Those defaults can be overridden:

```
PS C:\> Get-CimInstance -ComputerName localhost
SMBIOSBIOSVersion : 6.00
Manufacturer      : Phoenix Technologies LTD
Name              : PhoenixBIOS 4.0 Release 6.0
SerialNumber      : 56 4d e1 22 1b 63 b7
Version           : INTEL - 6040000
```

```
PSComputerName : localhost
```

You can also remove a specific default value:

```
PS C:\> $PSDefaultParameterValues.Remove("Get-CimInstance:ClassName")
PS C:\> $PSDefaultParameterValues
Name          Value
----          -----
Get-CimInstance:ComputerName LON-DC1
```

As with any variable, the contents of **\$PSDefaultParameterValues** is empty for each new Windows PowerShell session that you open.

You can read more about this technique by running **Help about\_parameters\_default\_values** in the shell.

## Demonstration: Setting Default Parameter Values

In this demonstration, you will see how to create and use default parameter values.

### Demonstration Steps

1. Create a default for the **-ComputerName** parameter of **Get-CimInstance**, by using the value **LON-DC1**.
2. Add a default for the **-ClassName** parameter of **Get-CimInstance** by using the value **Win32\_BIOS**.
3. Display the current default parameter values.
4. Retrieve the **Win32\_BIOS** CIM class from **LON-DC1**.
5. Retrieve the **Win32\_BIOS** CIM class from **localhost**.

Note: You will need remoting enabled for this to succeed if it is not already

6. Remove the default value for the **-ClassName** parameter of **Get-CimInstance**.
7. Display the current default parameter values.
8. Remove all default parameter values.

## Running External Commands

Windows PowerShell supports running external commands. Most of the time, the shell will internally create an instance of Cmd.exe, run the command, capture the output, and return the output to the Windows PowerShell pipeline as a collection of string object. For example:

```
PS C:\> ping localhost
Pinging LON-CL1 [::1] with 32 bytes of
data:
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Ping statistics for ::1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
```

- Most external commands run correctly in Windows PowerShell
- If a command's syntax uses Windows PowerShell special characters, it might not run correctly
- Use **--%** before the command's arguments to force the shell to run the command without trying to interpret the special characters

```
ICAcls.EXE --% C:\TEST /GRANT USERS:(F)
```

```
Approximate round-trip times in milli-seconds:  
Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

This technique works well for external commands that do not have a very complex command-line syntax. For more complex commands, Windows PowerShell may not correctly interpret the command and its arguments or parameters. In those cases, the command does not run correctly. This frequently happens when the command's syntax includes characters that have special meaning to Windows PowerShell. For example, the following command will not run correctly from inside Windows PowerShell:

```
ICACLS.EXE C:\TEST /GRANT USERS:(F)
```

Windows PowerShell 3.0 introduced a new syntax that lets you run commands such as this without error. Type the command name as usual, but precede its arguments by using `--%`. Windows PowerShell will not try to interpret anything after `--%` and will instead pass it along as-is to Cmd.exe. For example:

```
ICACLS.EXE --% C:\TEST /GRANT USERS:(F)
```

Remember that the `--%` operator is not available in earlier versions of Windows PowerShell. Also remember that after you use `--%`, Windows PowerShell will not examine the rest of the command. Variables, expressions, and other Windows PowerShell features will not be available. More details are available in [help\\_about\\_parsing](#).

## Lesson 2

# Creating Profile Scripts

In this lesson, you will learn about profile scripts. Profile scripts are a Windows PowerShell feature that enables you to specify a list of commands to run every time that you open a new shell session.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of a profile script
- List the correct locations for profile script storage
- Describe the security concerns of the profile script feature

### What is a Profile Script?

Profile scripts are a feature of Windows PowerShell host applications, not of the Windows PowerShell engine. In other words, the host application has the responsibility to implement profile scripts. Host applications provided by Microsoft®, such as the console host and the Integrated Scripting Environment (ISE) host, implement profile scripts. Other host applications may not.

A *profile script* is a regular Windows PowerShell script that is automatically loaded and run by the host application every time that the host application starts. Host applications may implement support for multiple profile scripts. This enables them to load and run several scripts every time that host application starts.

Profile scripts may contain any commands that you want. They are typically used to load modules, define aliases and variables, and perform other tasks that create a predefined shell environment. By using a profile script, you can make sure that your shell environment is always configured the way that you want it, every time you open a new session.

- Scripts that load and run automatically each time a new shell session is opened
- Implemented by the host application, not by the Windows PowerShell engine
  - Different hosts may define different profile script locations
  - Some hosts may not load profile scripts at all
- Use to define aliases, load modules, and configure the shell environment to meet your needs



**Note:** When you connect to a remote computer by using Windows PowerShell remoting, no profile script is run by default on the remote computer.

MCT USE ONLY. STUDENT USE PROHIBITED

## Profile Script Locations

The console and ISE host applications are each capable of loading up to four profile scripts every time that a new session is started. None of these script files exist by default, and not all the directory paths exist by default. Profile scripts are loaded in the order shown here, and are loaded only if the directory path and file exists when the shell session is started.

- Each host application defines the script files it will load and the order in which it loads them
- The console and ISE host applications share certain profile scripts, and define other scripts that are unique to each
- Easy to remember:  
**\Documents\WindowsPowerShell\profile.ps1**

 **Note:** In the following table, **\$home** refers to your user account home folder, for example, **C:\Users\Administrator**. The **Documents** folder may be **My Documents** on older versions of the Windows operating system. **\$PsHome** refers to the Windows PowerShell installation folder, for example, **C:\Windows\System32\WindowsPowerShell\v1.0**.

Profile Script Purpose	Console host application	ISE host application
Current user, current host	\$home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1	\$home\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
Current user, all hosts	\$home\Documents\WindowsPowerShell\Profile.ps1	\$home\Documents\WindowsPowerShell\Profile.ps1
All users, current host	\$pshome\Microsoft.PowerShell_profile.ps1	\$pshome\Microsoft.PowerShellISE_profile.ps1
All users, all hosts	\$pshome\Profile.ps1	\$pshome\Profile.ps1

As shown in this table, some profile scripts are shared by the console host and the ISE host. Other profile scripts are unique to each host. Some profile scripts are run for every user on the computer, but other profile scripts are unique to each user on the computer.

 **Best Practice:** If your goal is to have a consistent experience between both host applications, you should use **\$home\Documents\WindowsPowerShell\profile.ps1**.

## Profile Security Concerns

Remember that a profile script is just a plain text file that typically is located in your Documents folder. That means any process that is running on your computer while you are logged on can create and edit your profile scripts. A very simple piece of malware could create or change your profile script, and would not require elevated permissions.

However, you will usually open new Windows PowerShell sessions under elevated credentials so that your profile scripts will run under elevated credentials. This means that your profile script becomes an easy way for malware to run potentially dangerous commands under elevated credentials, without the malware itself having to have elevated credentials. The profile script is therefore able to bypass the "Principle of Least Privilege" or "Principle of Least Access" practiced by many organizations.

It is not the goal of Windows PowerShell to act as antimalware software. When malware is present on your computer, that malware can perform any number of dangerous tasks without using Windows PowerShell at all. However, the unique behavior of Windows PowerShell profile scripts makes them an attractive target.

There are two techniques that you can use to prevent your profile script from being used by malware:

- Always have up-to-date antimalware software installed. The best defense against malware is to keep it off your computer completely.
- Use the **AllSigned** Windows PowerShell execution policy, especially on computers that are used by administrators. Using this execution policy means that your script files, even your profile scripts, will have to be digitally signed. If a piece of malware changes a signed profile script, the signature on the script will be broken, and Windows PowerShell will display an error instead of running the script.

Remember that your first security goal should be to keep malware off your computer completely. If your environment is free of malware, your profile scripts do not represent any security risk. In organizations that practice extensive defense-in-depth techniques, the **AllSigned** execution policy is a small additional layer of security that adds more inconvenience to administrators, because the policy requires all scripts to be signed before they will run.

Another technique is available that can prevent profile scripts from becoming a risk. This technique is one that many organizations already use. In this technique, you log on to your computer by using a non-elevated user account. That account will have its own profile scripts, and you can leave those blank. When you run Windows PowerShell, open it by using a completely different user account that has elevated credentials. For example, you might log on to your computer as JeffP. When you run Windows PowerShell, you run it by using the user account AdminJeffP. The AdminJeffP account has a completely different set of profile scripts in completely different locations. Those scripts are unavailable to the non-elevated JeffP account, and so malware running as JeffP would be unable to insert commands into the profile scripts that are used by AdminJeffP. This approach provides good security, and does not create the inconvenience of the **AllSigned** execution policy. However, you should still try to keep malware out of your environment by using up-to-date anti-malware software.

- Profile scripts are local text files that can be modified by any process running under your user credentials, even non-elevated credentials
- But profile scripts typically run under elevated credentials, making them a potential security risk
- Be aware of the security risks
- To provide better security, use a completely different account to open Windows PowerShell sessions

## Demonstration: Creating a Profile Script

In this demonstration, you will see how to create a new profile script.

### Demonstration Steps

1. Open the Windows PowerShell ISE.
2. Set the local execution policy to **RemoteSigned**.
3. Create a new script.
4. Add a command that loads the ActiveDirectory module.
5. Add a command that changes to the **C:\** folder.
6. Save the script as the current-user, all-hosts profile script.
7. Close all open Windows PowerShell windows.
8. Open a new Windows PowerShell console by using the Administrator account.
9. Verify that the starting directory is **C:\**.
10. Verify that the ActiveDirectory module is loaded.

## Lesson 3

# Working with Alternative Credentials

In this lesson, you will learn how to create and use reusable objects for alternative credentials. Many Windows PowerShell commands accept alternative credentials, especially when you connect to remote computers. Reusable credential objects make it easier to specify a credential for multiple commands.

### Lesson Objectives

After completing this lesson, students will be able to:

- Explain the purpose of a credential
- Create and use a credential
- Describe the security concerns of persisting a credential object

### What Is a Credential?

Many Windows PowerShell commands accept alternative credentials. That means these commands enable you to specify a credential other than the one that you used to open the Windows PowerShell session. Alternative credentials are typically used when you connect to a remote computer, when your local credentials do not have permission to complete a particular task on the remote computer. Commands that accept alternative credentials do so by using a **-Credential** parameter.

- Provide alternate credentials to commands that have a **-Credential** parameter
- Allows the command to perform its task using that credential rather than the one you used to open the shell
- You will be prompted for the password by means of a dialog box
- You can also create a reusable credential object that includes both the user name and the password

The easiest way to specify an alternative credential is to provide a user name, or a domain name and user name, to the **-Credential** parameter. For example:

```
Get-Service -Name * -ComputerName LON-DC1 -Credential ADATUM\Administrator
```

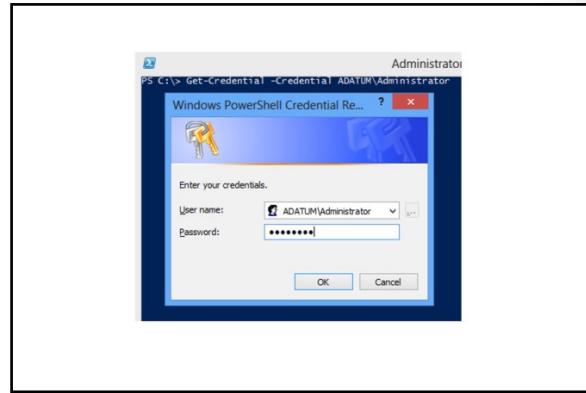
When you specify a user name, Windows PowerShell uses a graphical dialog box to prompt you for the account password. You cannot directly specify the password on the command-line. Doing this would let you list a password in clear text, and that would be a security issue.

If you have to use the same credential multiple times, having to repeatedly type the same password can be time-consuming, error-prone, and not what you want. Therefore, the shell also lets you create a reusable credential object that contains both the user name and an encrypted password.

## Creating and Using a Credential

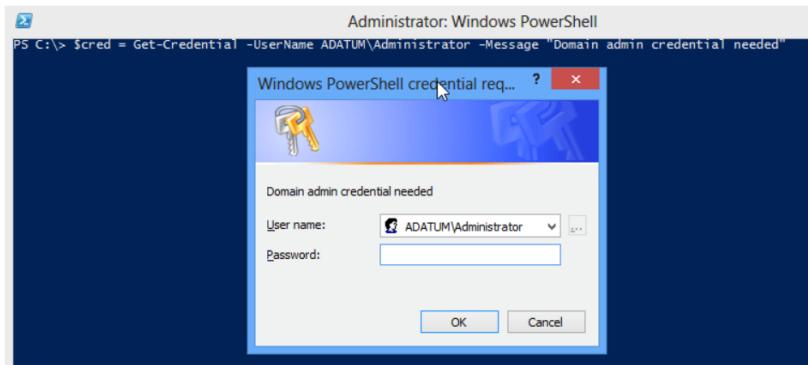
To create a reusable credential object, use the **Get-Credential** command. You can use the **-Credential** parameter to specify a user name, although you can exclude this parameter to have the user enter the desired user name in the dialog box instead. You can also use the **-Message** parameter to provide a description of what the credential will be used for, or what credential is needed. If you do not specify **-Credential**, you must specify **-Message**. When you specify **-Message**, you can also specify **-UserName** to provide a default user name. For example:

```
$cred = Get-Credential -UserName ADATUM\Administrator -Message "Domain admin credential needed"
```



This command produces the dialog box shown here:

The **Get-Credential** command displays a dialog box to collect the user name and password.



**FIGURE 12.1: PROMPTING FOR A CREDENTIAL BY USING A DIALOG BOX.**

The results of **Get-Credential** are typically stored in a variable. That variable can then be provided to the **-Credential** parameter of other commands. For example:

```
Get-Service -Name * -ComputerName LON-DC1 -Credential $cred
```



**Note:** The **Get-Credential** command does not validate the credential you provide. The credential is not validated until you attempt to use it as part of another command.

As with all variables, **\$cred** in this example will exist only until the shell session closed. If you have a credential that you use regularly, consider adding the **Get-Credential** command to a profile script. If you do this, you are prompted for the credential every time that a new shell window opens, and the credential variable will be available for use during that shell session.

## Persisting Credentials

Many Windows PowerShell users ask whether you can save a credential object to disk. They typically have a goal of using a credential without being prompted for its password. There are many ways to achieve this goal. However, they all include significant security risks.

One approach is to save the password in a file. For example, this command will prompt for a password and then save it to the file C:\password.txt:

```
read-host "Password" -assecurestring |  
convertfrom-securestring | out-file C:\password.txt
```

- It is *possible* to store a password to disk in an encrypted form
- That password can be read from disk and used to create a credential object, without prompting for the credential password
- However, this approach is *not* recommended because the password is not stored in a form that is protected from accidental discovery or disclosure
- Stored passwords are compromised passwords

This command will then read the password from the file, and create a new credential object that uses the password:

```
$pass = get-content C:\password.txt | convertto-securestring  
$cred = new-object -type System.Management.Automation.PSCredential -arg  
"ADATUM\Administrator", $pass
```

The problem with this approach, and with other similar approaches, is that anyone could read the password from the text file and create the same credential object. Although the file has no reference to the user account that the password is for, the password is nevertheless not stored in a secure manner.

As a best practice, you should avoid storing a password to disk under any circumstances, for the same reasons that you should avoid writing a password down on a piece of paper and attaching it to your computer monitor. Stored passwords are too easily discovered and compromised.

## Demonstration: Creating and Using a Credential

In this demonstration, you will see how to create and use a credential object.

### Demonstration Steps

1. Create a credential object for **ADATUM\Administrator**. Store the credential in the variable **\$cred**.
2. Use the credential object to query the 10 newest Security event log entries from LON-DC1.

# Lab: Practicing Advanced Techniques

## Scenario

You have to practice how to use several advanced Windows PowerShell features. This includes date and string manipulation, advanced comparison operators, and alternative credentials. You also have to create a profile script.

## Objectives

After completing this lab, students will be able to:

- Use advanced Windows PowerShell techniques
- Create and use alternative credentials
- Create a profile script

## Lab Setup

Estimated Time: 75 minutes

Virtual Machines: 10961B-LON-DC1, 10961B-LON-CL1

User Name: ADATUM\Administrator

Password: Pa\$\$w0rd

The changes that you make during this lab will be lost if you revert your virtual machines at another time during class.

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

1. On the host computer, move the pointer over the bottom left corner of the taskbar, click **Start**, and then click **Hyper-V Manager** on the Start screen.
2. In Hyper-V® Manager, click 10961B-LON-DC1, and in the Actions pane, click **Start**.
3. In the Actions pane, click **Connect**. Wait until the virtual machine starts.
4. Sign in by using the following credentials:
  - User name: **Administrator**
  - Password: **Pa\$\$w0rd**
  - Domain: **ADATUM**
5. Repeat steps 2 through 4 for 10961B-LON-CL1.
6. The lab steps should be performed on the 10961B-LON-CL1 virtual machine.

## Exercise 1: Using Advanced Techniques

### Scenario

In this exercise, you will practice how to use several advanced Windows PowerShell techniques.

The main tasks for this exercise are as follows:

1. Use Advanced Operators
2. Manipulate string values

MCT USE ONLY. STUDENT USE PROHIBITED

3. Manipulate date values

► **Task 1: Use Advanced Operators**

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. To prepare the environment for this lab run:

```
. E:\Mod12\Labfiles\Lab12.ps1
```

Notice that there are a period and a space before the **E:\** in the file name you may also need to modify the execution policy depending on your setting.

The script that you ran in step 2 created several variables and populated them with data. You will use those variables to answer several questions and to perform several tasks.

For each of the following, write a single command that will answer the question or perform the task:

3. Does **\$ServiceNames** include the value **WinRM**?
4. Does **\$ServiceNames** include the value **Spooler**?
5. What is the second value in the array **\$ComputerNames**?
6. Produce a comma-separated string that contains the values in the array **\$ComputerNames**.
7. Produce a tab-delimited string that contains the values in **\$ComputerNames**.
8. In the variable **\$phrase**, replace the word **dog** with the word **gelding**.
9. **\$List** contains a comma-separated list. Display the second value in this list.

► **Task 2: Manipulate string values**

1. If you have not already done so, in Task 1, run: **. E:\Mod12\Labfiles\Lab12.ps1**. Notice that there are a period and a space before the **E:\** in the file name.

The script that you ran in step 1 created several variables and populated them with data. You will use those variables to answer several questions and to perform several tasks.

For each of the following, write a single command that will complete the task:

2. Display an all-uppercase version of the contents of **\$phrase**.
3. Display an all-lowercase version of the contents of **\$phrase**.
4. In **\$phrase**, replace the value **over** with the value **around**, and display an all-uppercase version of the result.
5. Display the sixth through the eighth characters in **\$List**.
6. Display the contents of **\$padded** so that there are no additional spaces before or after **value**.
7. Display the number of characters in **\$phrase**.
8. Display **True** or **False**, depending on whether the contents of **\$phrase** starts with the value **The**.
9. Display the contents of **\$unpadded** so that 10 additional spaces are added before **value**.

► **Task 3: Manipulate date values**

1. Store the current date in the variable **\$today**.

For each of the following, write a single command that uses **\$today** to complete the task:

2. Display the full date of 10 days ago.
3. Display the full date of 30 days from today.

4. Display the current hour.
5. Display the current month.
6. Display today's date as a short date string.
7. Display today's date in Universal time.
8. Display the current time as a long time string.

**Results:** After completing this exercise, you will have practice with using several Windows PowerShell techniques.

## Exercise 2: Using Alternative Credentials

### Scenario

In this exercise, you will practice by using alternative credentials.

The main tasks for this exercise are as follows:

1. Create a credential object
2. Run a Command By Using a Credential

► **Task 1: Create a credential object**

- Create a new credential object for **ADATUM\Administrator**. Store the credential in the variable **\$admin**.

► **Task 2: Run a Command By Using a Credential**

1. Using **\$admin** and **Invoke-Command**, tell LON-DC1 to produce a list of all Active Directory user accounts and store the list as a CSV file on the server.
2. Save the CSV file as **C:\Users.csv** on the server and verify the file is created successfully.

**Results:** After completing this exercise, you will have practiced how to use alternative credentials.

## Exercise 3: Create a Profile Script

### Scenario

In this exercise, you will create a profile script.

The main tasks for this exercise are as follows:

1. Create a Profile Script
2. Define default values for a command
3. Test the profile script
4. To prepare for another module

► **Task 1: Create a Profile Script**

1. On the LON-CL1 virtual machine, if you do not have a Windows PowerShell ISE window open, open one.
2. In the ISE, open a new script.

3. In the new script, enter a command to change to the **C:\** drive.
4. In the new script, add a command that creates a new credential object for **ADATUM\Administrator**, storing the credential in **\$cred**.
5. Save the script as the current-user, all-hosts profile script.

► **Task 2: Define default values for a command**

1. In your profile script, add a command that sets default values for a command:
  - o Command: **Get-EventLog**
  - o Parameter: **-LogName**
  - o Value: **Security**
  - o Command: **Get-EventLog**
  - o Parameter: **-Newest**
  - o Value: **10**
2. Save the profile script.

► **Task 3: Test the profile script**

1. Open a new Windows PowerShell console window.
2. Verify that the shell is in the **C:\** folder.
3. Verify the contents of **\$cred**.
4. Run **Get-EventLog** without any parameters and verify that 10 entries from the Security log are shown.

► **Task 4: To prepare for another module**

If you are intending continuing to another module to perform labs, once you have finished this lab, revert the virtual machines to their initial state. To do this, perform the following steps:

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

**Results:** After completing this exercise, you will have created a profile script.

**Question:** If your user profile is redirected to a network location, will profile scripts still work?

**Question:** How can you quickly obtain a list of methods and properties for a string object or for a date object?

## Module Review and Takeaways

### Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Profile script will not load and run.	

### Real-world Issues and Scenarios

Some administrators have their profile script load such a large number of add-ins and modules that Windows PowerShell can take several minutes to start. Try to include only those add-ins or modules that you absolutely need constantly. Also, remember that some add-ins and modules can conflict with one another. If you experience strange behavior in the shell, try starting a new shell session by running **powershell.exe –noprofile** to suspend profile script execution. Then, load only one add-in or module at a time to see which one caused the strange behavior.

## Course Evaluation

Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

### Course Evaluation

- Your evaluation of this course will help Microsoft understand the quality of your learning experience.
- Please work with your training provider to access the course evaluation form.
- Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 1: Getting Started with Windows PowerShell

## Lab A: Configuring Windows PowerShell

### Exercise 1: Configure the Windows PowerShell Console Application

► **Task 1: Start the 64-bit console application as Administrator and pin Windows PowerShell icon to the taskbar**

NOTE: There are multiple keyboard languages installed to support course delivery globally. As such some key strokes may give different results than expected if a language different to what you expect is set as the input method. In the **10961B-LON-CL1** virtual machine the default keyboard language is set to **ENG (US)**. If you require a different keyboard language you should click on the system icon in the bottom right hand corner, which indicates the keyboard language, and select the appropriate language for your region. Alternatively you can use the following short cut keys sequence, hold down the Left ALT+ press Left SHIFT one key stroke at a time, while continuing to hold down the Left ALT key, to scroll through the languages or by pressing Windows Key + Space to select your required language.

1. On the 10961B-LON-CL1 virtual machine, log on as **Adatum\Administrator**.
2. Press the Windows key on your keyboard.
3. Type **powersh** to display the **Windows PowerShell** icon.
4. Make sure that the icon caption says **Windows PowerShell** and that it does not say **Windows PowerShell (x86)**. Right-click the icon, and then click **Run as administrator**.
5. Make sure that the window title bar says **Administrator** and does not include the text **(x86)**. This indicates that it is the 64-bit console application and it is being run as Administrator.
6. On the taskbar, right-click the **Windows PowerShell** icon and then click **Pin this program to taskbar**. 64-bit Windows PowerShell should now be open, running as Administrator, and available on the taskbar for future use.

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

1. To configure the shell to use the Consolas font:
  - a. Click the control box in the upper-left corner of the Windows PowerShell console window.
  - b. Click **Properties**.
  - c. In the Windows PowerShell Properties dialog box on the **Font** tab, select **Consolas**. Select a font size that is comfortable to read.
2. To select alternate display colors:
  - On the **Colors** tab, review the available foreground (text) and background colors.
3. To size the window and to remove the horizontal scroll bar:
  1. On the **Layout** tab, in the **Window Size** settings, change the area's **Width** and **Height** values until the Windows PowerShell console pane preview fits completely within the Window Preview area.
  2. On the **Layout** tab in the **Screen Buffer Size** settings, change the **Width** value to be the same as the **Windows Size** settings **Width** value.
  3. Click **OK**. The console application should now be ready for use.

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

1. To start a transcript in the Windows PowerShell console, run:

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

You have now started a transcript, and it will save all of your commands and command output until you run **Stop-Transcript** or until you close the shell window.

2. You can view all the commands you have run and the command output by opening the file C:\DayOne.txt.

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

## Exercise 2: Configure the Windows PowerShell ISE Application

### ► Task 1: Open the 64-bit Windows PowerShell ISE application as Administrator

1. Ensure you are still logged onto the 10961B-LON-CL1 virtual machine as Adatum\Administrator.
2. Do one of the following:

In the Windows PowerShell console application, type **ise** and then press Enter. Notice that this will work correctly only when the console is running as Administrator.

or

Right-click the **Windows PowerShell** icon on the taskbar and then click **Run ISE as Administrator**.

3. You should now have a 64-bit ISE application running as Administrator.

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

1. To configure the ISE to use a single-pane view:
  - a. On the Windows PowerShell ISE toolbar, click the **Show Script Pane Maximized** button (third from the right).
  - b. Click the blue **Hide Script Pane** up-arrow icon on the extreme right side until the console pane is displayed.
2. Use the **Show Command Add-on** button (rightmost button on the toolbar) to hide or view the Command Pane.
3. To adjust the font size, use the slider in the lower-right corner of the window to adjust the font size until you can read it comfortably.

**Results:** After completing this lab, you will have customized the appearance of the Windows PowerShell ISE application.

## Lab B: Finding and Running Basic Commands

### Exercise 1: Finding Commands

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

1. Press the Windows logo key, type **PowerShell**, right click the Windows PowerShell tile, and click **Run as administrator**.
2. Run:

```
help *resolve*
```

or:

```
Get-Command *resolve*
```

These display a list of commands that use *Resolve* in their names. This should lead you to the **Resolve-DNSName** command.

3. Run:

```
help *adapter*
```

or:

```
Get-Command *adapter*
```

These display a list of commands that use *Adapter* in their names. This should lead you to the **Set-NetAdapter** command. Then, run **help set-netadapter** to view the Help for that command. This should lead you to the **-MACAddress** parameter.

4. Run:

```
help *sched*
```

or:

```
Get-Command *sched*
```

These display a list of commands that use *Sched* in their name. This should lead you to the **Enable-ScheduledTask** command (you may also find the **Enable-PSScheduledTask** command that is similar).

5. Run:

```
Get-Command -Verb Block
```

Or:

```
help *block*
```

These display a list of commands. This should lead you to the **Block-SMBShare** command. Then, run **help block-smbshare** to learn that the command applies a Deny entry to the file share Discretionary Access Control List (DACL).

6. Run:

```
help *branch*
```

This will cause the Help system to conduct a full-text search, because no commands use *branch* in their names. Or, run:

```
help *cache*
```

or:

```
Get-Command *cache*
```

These will display a list of commands. Either way, you should discover the **Clear-BCCache** command.

7. Run any of the following:

```
help *firewall*
```

or

```
Get-Command *firewall*
```

or

```
help *rule*
```

or

```
Get-Command *rule*
```

These display a list of commands that use those keywords in their names. This should lead you to the **Get-NetFirewallRule** command

Then, run:

```
help get-netfirewallrule -full
```

This will display the Help for the command. This should let you discover the **-Enabled** parameter.

8. Run:

```
help *address*
```

This will display a list of commands that use *address* in their names. This should lead you to the **Get-NetIPAddress** command.

9. Run:

```
help *suspend*
```

or:

```
Get-Command -verb suspend
```

These display a list of commands. This should lead you to the **Suspend-PrintJob** command.

10. Run:

```
help *format*
```

or:

```
Get-Command -Verb format
```

These display a list of commands. This should lead you to the **Format-Volume** command.

**Results:** After completing this exercise, you will have demonstrated your ability to use the command discoverability features of Windows PowerShell™ to find new commands that perform specific tasks.

## Exercise 2: Finding and Running Commands

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

1. Ensure you are working on the 10961B-LON-CL1 virtual machine logged on as Adatum\Administrator.
2. To display a list of enabled firewall rules, run:

```
Get-NetFirewallRule -Enabled True
```

3. To display a list of all local IPv4 addresses, run

```
Get-NetIPAddress -AddressFamily IPv4
```

4. To set the startup type of the BITS service, run :

```
Set-Service -Name BITS -StartupType Automatic
```

5. To test the connection to LON-DC1, run:

```
Test-Connection -ComputerName LON-DC1 -Quiet
```

Notice that this command returns only a True or False value, without any other output.

6. To display the newest 10 entries from the Security event log, run:

```
Get-EventLog -LogName Security -Newest 10
```

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

## Exercise 3: Using "About" Files

### ► Task 1: Locate and read "About" Help files

1. Ensure you are still on the 10961B-LON-CL1 virtual machine logged on as Adatum\Administrator from the previous exercise.
2. To find operators used for wildcard string comparison, run:

```
help *comparison*
```

then run:

```
help about_comparison_operators -ShowWindow.
```

Notice the **-Like** operator in the **about\_Comparison\_Operators**. To find it, in the **Find** text box, type **wild**, and then click **Next**.

3. After reading the **about\_Comparison\_Operators** file, you should learn that typical operators are not case-sensitive. Specific case-sensitive operators are provided in **about\_Comparison\_Operators**.
4. To display the **COMPUTERNAME** environment variable, run:

```
$env:computername
```

You could read about this technique in **about\_environment\_variables**.

5. Run:

```
help *signing*
```

then run:

```
help about_signing
```

Then read about code signing. You should learn that **Makecert.exe** is used to create a self-signed digital certificate.

### ► Task 2: To prepare for the next module

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

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

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

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 2: Working with the Pipeline

## Lab A: Using the Pipeline

### Exercise 1: Selecting and Sorting Data

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

1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
2. To find a command that can display the current date, run:

```
help *date*
```

Notice the **Get-Date** command.

3. To display the members of the date object, run:

```
Get-Date | Get-Member
```

Notice the **DayOfYear** property.

4. To display only the day of the year, run:

```
Get-Date | Select-Object -Property DayOfYear
```

#### ► Task 2: Display information about installed hotfixes.

1. To find a command that can display a list of hotfixes, run:

```
help *hotfix*
```

Notice the **Get-Hotfix** command.

2. Run:

```
Get-Hotfix | Get-Member
```

This will display the properties of the hotfix object. If needed, run **Get-Hotfix** to see some of the values that typically appear in those properties.

3. To display a list of installed hotfixes, including only specified properties, run:

```
Get-Hotfix | Select-Object -Property HotFixID, InstalledOn, InstalledBy
```

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

1. To find a command that can display DHCP scopes, run:

```
help *scope*
```

Notice the **Get-DHCPServerv4Scope** command.

2. To display the command Help, run:

```
Help Get-DHCPServerv4Scope -ShowWindow
```

Notice the available parameters.

3. To display a list of scopes, run:

```
Get-DHCPServerv4Scope -ComputerName LON-DC1
```

4. To display a list of scopes that includes only the specified properties, run:

```
Get-DHCPServerv4Scope -ComputerName LON-DC1 | Select-Object -Property ScopeId,SubnetMask,Name
```

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

1. To find a command that can display firewall rules, run:

```
help *rule*
```

Notice the **Get-NetFirewallRule** command.

2. To display a list of firewall rules, run:

```
Get-NetFirewallRule
```

3. To read command Help, run:

```
Help Get-NetFirewallRule -ShowWindow
```

4. To display a list of enabled rules, run:

```
Get-NetFirewallRule -Enabled True
```

5. To display a list of enabled rules that includes only specified properties, run:

```
Get-NetFirewallRule -Enabled True | Select-Object -Property DisplayName,Profile,Direction,Action | Sort-Object -Property DisplayName
```

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

1. To find a command that can display network neighbors, run:

```
help *neighbor*
```

Notice the **Get-NetNeighbor** command.

2. To read command Help, run:

```
Help Get-NetNeighbor -ShowWindow
```

3. To display the neighbor list, run:

```
Get-NetNeighbor
```

4. To display a sorted neighbor list, run:

```
Get-NetNeighbor | Sort-Object -Property State
```

5. To display a sorted neighbor list that includes only specified properties, run:

```
Get-NetNeighbor | Sort-Object -Property State | Select-Object -Property IPAddress,State
```

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

1. To ping LON-DC1, run:

```
ping LON-DC1
```

2. To ping LON-CL1, run:

```
ping LON-CL1
```

3. To display a list of commands that use the word *cache*, run:

```
help *cache*
```

Notice the **Get-DnsClientCache** command.

4. To display the DNS cache, run:

```
Get-DnsClientCache
```

5. To display the DNS cache and include only specified properties, run:

```
Get-DnsClientCache | Sort Name | Select Name,Type,TimeToLive
```

6. You will notice that the "Type" data doesn't return what you expect and is raw numerical data i.e. A, CNAME etc however each number does map directly to a "Record Type" and you can filter for such i.e. 1=A, 5=CNAME etc and you will learn how to add additional filters later in the module to determine the numbers and corresponding record types. i.e. **Get-DnsClientCache | Where-Object type -eq 5** returns only CNAME records. You will notice a similar situation for other data returned such as "Status".

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

## Lab B: Converting, Exporting, and Importing Objects

### Exercise 1: Converting Objects

► **Task 1: Produce an HTML report listing the processes running on a computer**

 **Note:** In this document, long commands are typically displayed on several lines. Doing so helps prevent an unintended line break in the middle of a command. However, when you type these commands, you should type them as a single line. That line may wrap on your screen into multiple lines, but the command will still work. You should press Enter only after typing the entire command.

1. Log on to the 10961B-LON-CL1 virtual machine logged in as **Adatum\Administrator**.

2. To display a list of running processes, run:

```
Get-Process
```

3. To display a list of running processes, sorted in reverse alphabetic order by process name, that shows only the process name, ID, virtual memory, and physical memory consumption, run:

```
Get-Process | Sort Name -Descending | Select Name, ID, VM, PM
```

4. To view the Help for **ConvertTo-HTML**, run:

```
help ConvertTo-HTML -ShowWindow
```

5. To convert the process list to an HTML page, run:

```
Get-Process | Sort Name -Descending | ConvertTo-HTML -Property Name, ID, VM, PM
```

6. To save the HTML page in a file, run:

```
Get-Process |  
Sort Name -Descending | ConvertTo-HTML -Property Name, ID, VM, PM |  
Out-File ProcReport.html
```

7. To view the HTML file, run:

```
Invoke-Expression .\ProcReport.html
```

8. To create the modified HTML file, run:

```
Get-Process |  
Sort Name -Descending | ConvertTo-HTML -Property Name, ID, VM, PM -PreContent  
"Processes" -PostContent (Get-Date) |  
Out-File ProcReport.html
```

9. To display the HTML file, run:

```
Invoke-Expression .\ProcReport.html
```

**Results:** After completing this exercise, you will have converted objects to different forms of data.

## Exercise 2: Importing and Exporting Objects

- **Task 1: Create a comma-separated values (CSV) file listing the most recent 10 entries from the System event log.**

 **Note:** When typing these commands, you should type them as a single line, and press Enter only once, after typing the entire line. However, in the Console application, you can also type these commands exactly as they are shown. Typically, that means pressing Enter after each vertical pipe (|) character. If you use this technique, you will have to press Enter on a blank line, after typing all of the lines, to execute the command.

1. To display the System event log, run:

```
Get-EventLog -Newest 10 -LogName System
```

2. To convert the log to CSV, run:

```
Get-EventLog -Newest 10 -LogName System |  
ConvertTo-Csv
```

3. To export the log as a CSV file, run:

```
Get-EventLog -Newest 10 -LogName System |  
Export-Csv SysEvents.csv
```

4. To view the CSV file, run:

```
Notepad SysEvents.csv
```

5. To export the log and remove the comment line containing type information, run:

```
Get-EventLog -Newest 10 -LogName System |  
Export-Csv SysEvents.csv -NoTypeInformation
```

6. To view the revised CSV file, run:

```
Notepad SysEvents.csv
```

- **Task 2: Create an XML file listing services**

1. To display a list of services that shows stopped services last, run:

```
Get-Service |  
Sort Status -Descending
```

2. To export the service list to an XML file, run:

```
Get-Service |  
Sort Status -Descending |  
Export-CliXML Services.xml
```

3. To display the XML file, run:

```
Notepad Services.xml
```

4. To choose specified columns to be included in the file, run:

```
Get-Service |  
Sort Status -Descending |  
Select Name,DisplayName,Status | Export-CliXML Services.xml
```

5. To display the revised file, run:

```
Notepad Services.xml
```

► **Task 3: Produce a pipe-delimited list of the most recent 20 Security event log entries**

1. To view the Help file for **ConvertTo-HTML**, run:

```
Help ConvertTo-CSV -ShowWindow
```

2. To display the log entries, run:

```
Get-EventLog -newest 20 -LogName Security
```

3. To display only specified properties of the log entries, run:

```
Get-EventLog -newest 20 -LogName Security |  
Select EventID,TimeWritten,Message
```

4. To export the log entry list as a pipe-delimited file, run:

```
Get-EventLog -newest 20 -LogName Security |  
Select EventID,TimeWritten,Message |  
Export-Csv Security.pdd -Delimiter '|'
```

5. To display the file, run:

```
Notepad Security.pdd
```

► **Task 4: Import data from a pipe-delimited file**

- To import data from the pipe-delimited file, run:

```
Import-Csv Security.pdd -Delimiter '|' |  
Select -First 10
```

**Results:** After completing this lab, you will have imported data from and exported data to external storage.

## Lab C: Filtering Objects

### Exercise 1: Filtering Objects

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

1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
2. To find a command that can list Active Directory® users, run:

```
help *user*
```

Notice the **Get-ADUser** command.

3. To view the Help for the command, run:

```
help Get-ADUser -ShowWindow
```

Notice that the **-Filter** parameter is mandatory. Review the examples for the command.

4. To display a list of all users, run:

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

5. To display a list of all users in a specified container, run:

```
Get-ADUser -Filter * -SearchBase "cn=Users,dc=Adatum,dc=com"
```

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

1. To display a list of Security event log entries that have the event ID 4624, run:

```
Get-EventLog -LogName Security |  
Where EventID -eq 4624
```

2. To display the same list, showing only specified properties, run:

```
Get-EventLog -LogName Security |  
Where EventID -eq 4624 |  
Select TimeWritten,EventID,Message
```

3. To convert the list to an HTML file, run:

```
Get-EventLog -LogName Security |  
Where EventID -eq 4624 |  
Select TimeWritten,EventID,Message |  
ConvertTo-HTML |  
Out-File EventReport.html
```

4. To view the HTML file, run:

```
Invoke-Expression .\EventReport.html
```

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

1. To display a directory listing of all items in the **CERT:** drive, run:

```
Get-ChildItem -Path CERT: -Recurse
```

2. To display the members of the objects, run:

```
Get-ChildItem -Path CERT: -Recurse |  
Get-Member
```

3. To show only the certificates that do not have a private key, run either this:

```
Get-ChildItem -Path CERT: -Recurse |  
Where HasPrivateKey -eq $False
```

or this:

```
Get-ChildItem -Path CERT: -Recurse |
```

```
Where { $PSItem.HasPrivateKey -eq $False }
```

4. To display the list again by using the specified filtering criteria, run:

```
Get-ChildItem -Path CERT: -Recurse |  
Where { $PSItem.HasPrivateKey -eq $False -and $PSItem.NotAfter -gt (Get-Date) -and  
$PSItem.NotBefore -lt (Get-Date) }
```

5. To display the list again by using the specified filtering criteria and showing only the specified properties, run:

```
Get-ChildItem -Path CERT: -Recurse |  
Where { $PSItem.HasPrivateKey -eq $False -and $PSItem.NotAfter -gt (Get-Date) -and  
$PSItem.NotBefore -lt (Get-Date) } |  
Select Issuer,NotBefore,NotAfter
```

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

1. To display a list of disk volumes, run:

```
Get-Volume
```

If you did not know the command name, you could have run **Help \*volume\*** to discover the command name.

2. To display a list of object members, run:

```
Get-Volume | Get-Member
```

Notice the **SizeRemaining** property.

3. To display only volumes that have more than zero bytes of free space, run:

```
Get-Volume |  
Where-Object { $PSItem.SizeRemaining -gt 0 }
```

4. To display only volumes that have less than 99 percent free space, and more than zero bytes of free space, run:

```
Get-Volume |  
Where-Object { $PSItem.SizeRemaining -gt 0 -and $PSItem.SizeRemaining / $PSItem.Size  
-lt .99 }
```

5. To display only volumes that have less than 10 percent free space and more than zero bytes of free space, run:

```
Get-Volume |  
Where-Object { $PSItem.SizeRemaining -gt 0 -and $PSItem.SizeRemaining / $PSItem.Size  
-lt .1 }
```

This command may not produce any output on your lab computer if the computer has more than 10 percent free space on all of its volumes.

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

1. To find a command that can display Control Panel items, run:

```
help *control*
```

Notice the **Get-ControlPanelItem** command.

- To display a list of Control Panel items, run:

```
Get-ControlPanelItem
```

- To display items in the **System and Security** category, run:

```
Get-ControlPanelItem -Category 'System and Security'
```

Notice that you do not have to use **Where-Object**.

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

## Lab D: Enumerating Objects

### Exercise 1: Enumerating Objects

- **Task 1: Display a list of key algorithms for all encryption certificates installed on your computer.**

- To display a list of all items in the **CERT:** drive, run:

```
Get-ChildItem -Path CERT: -Recurse
```

- To display the members of those objects, run:

```
Get-ChildItem -Path CERT: -Recurse | Get-Member
```

Notice the **GetKeyAlgorithm()** method in the list.

- To display a list of key algorithms for each installed certificate, run:

```
Get-ChildItem -Path CERT: -Recurse |
ForEach GetKeyAlgorithm
```

- To display the same list by using **Select-Object**, run:

```
Get-ChildItem -Path CERT: -Recurse |
Select Issuer,@{n='KeyAlgorithm';e={$PSIItem.GetKeyAlgorithm()}}
```

- **Task 2: Use enumeration to produce 100 random numbers**

- To find a command that can produce random numbers, run:

```
help *random*
```

Notice the **Get-Random** command.

- To view the Help for the command, run:

```
help Get-Random -ShowWindow
```

Notice the **-SetSeed** parameter.

- To place 100 numeric objects into the pipeline, run:

```
1..100
```

4. To produce 100 random numbers, run:

```
1..100 |  
ForEach { Get-Random -SetSeed $PSItem }
```

► **Task 3: Execute a method of a Windows Management Instrumentation (WMI) object**

1. Close all applications other than the Windows PowerShell™ console.

2. Run:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges
```

3. To display the members of the object, run:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges |  
Get-Member
```

4. Notice the **Reboot()** method.

5. To restart the computer, run:

```
Get-WmiObject -Class Win32_OperatingSystem -EnableAllPrivileges |  
ForEach Reboot
```

► **Task 4: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have written commands that manipulate multiple objects in the pipeline.

# Module 3: Understanding How the Pipeline Works

## Lab: Working with Pipeline Parameter Binding

### Exercise 1: Predicting Pipeline Behavior

#### ► Task 1: Review existing commands

For these tasks, you may run individual commands and **Get-Member** to see what kinds of objects the commands produce. You may also view the Help for any of these commands.

However, do not run the whole command shown. If you do run the whole command, it may produce an error. The error does not mean the command is written incorrectly.

1. This command is intended to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Get-Service -Name *
```

Will the command achieve the goal?

No. **Get-Service** does not accept **ADComputer** objects from the pipeline.

2. This command is intended to list the services that are running on every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-Service -Name *
```

Will the command achieve the goal?

Yes. The **-ComputerName** parameter accepts pipeline input by using **ByPropertyName**.

3. This command is intended to query an object from every computer in the domain:

```
Get-ADComputer -Filter * | Select @{n='ComputerName';e={$PSItem.Name}} | Get-WmiObject -Class Win32_BIOS
```

Will the command achieve the goal?

No. The **-ComputerName** parameter of **Get-WmiObject** does not accept pipeline input.

4. The file **Names.txt** lists one computer name per line.

This command is intended to list the services that are running on every computer that is listed in **Names.txt**.

```
Get-Content Names.txt | Get-Service
```

Will the command achieve the goal?

No. Pipeline input of the type **String** will attach to the **-Name** parameter of **Get-Service** by using **ByValue**. The **-ComputerName** parameter does not accept pipeline input by using **ByValue**.

5. The file **Names.txt** lists one computer name per line.

This command is intended to list the services that are running on every computer that is listed in **Names.txt**.

```
Get-Service -ComputerName (Get-Content Names.txt)
```

Will the command achieve the goal?

Yes. **Get-Content** produces objects of the type **String**, and the **-ComputerName** parameter can accept those objects.

6. This command is intended to list the services that are running on every computer in the domain:

```
Get-Service -ComputerName (Get-ADComputer -Filter *)
```

Will the command achieve the goal?

No. The **-ComputerName** parameter cannot accept objects of the type **ADComputer**.

7. This command is intended to list the Security event log entries from every computer in the domain:

```
Get-EventLog -LogName Security -ComputerName (Get-ADComputer -Filter * | Select -Expand Name)
```

Will the command achieve the goal?

Yes. The parenthetical command produces objects of the type **String** that are computer names. Those are accepted by the **-ComputerName** parameter.

## ► Task 2: Write new commands that perform specified tasks

In each of these tasks, you are asked to write a command that achieves a specified goal. *Do not run these commands.* Write them on paper.

You may run individual commands and pipe their output to **Get-Member** to see what objects those commands produce. You may also read the Help for any command.

1. Write a command displays the most recent 50 System event log entries from each computer in the domain.

Because **Get-EventLog** does not accept pipeline input for its **-ComputerName** parameter, you have to write this command:

```
Get-EventLog -LogName System -Newest 50 -ComputerName (Get-ADComputer -filter * | Select-Object -ExpandProperty Name)
```

2. You have a text file that is named **Names.txt** that contains one computer name per line. Write a command that uses **Restart-Computer** to restart each computer that is listed in the file. Do not use a parenthetical command.

Because the **-ComputerName** parameter of **Restart-Computer** accepts pipeline input by using **ByValue**, the following command will achieve the goal:

```
Get-Content Names.txt | Restart-Computer
```

3. You have a file that is named **Names.txt** that contains one computer name per line. Write a command that uses **Test-Connection** to test the connectivity to each computer that is listed in the file.

The **-ComputerName** parameter of **Test-Connection** accepts pipeline input only by using **ByPropertyName**. The objects produced by **Get-Content** do not have a property named **ComputerName**, nor do they have an existing property that can be renamed to **ComputerName**. Therefore, you must use a parenthetical command:

```
Test-Connection -ComputerName (Get-Content Names.txt)
```

4. Write a command that uses **Set-Service** to set the start type of the **WinRM** service to **Auto** on every computer in the domain. Do not use a parenthetical command.

Because the **-ComputerName** parameter of **Set-Service** accepts pipeline input by using **ByPropertyName**, you can write the following command:

```
Get-ADComputer -filter * |  
Select-Object @{n='ComputerName';e={$PSItem.Name}} |  
Set-Service -Name WinRM -StartupType Auto
```

You must use **Select-Object** because the **ADComputer** objects have a **Name** property, not a **ComputerName** property.

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have reviewed and written several Windows PowerShell™ commands.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 4: Using PSProviders and PSDrives

# Lab: Using PSProviders and PSDrives

### Exercise 1: Create a New Folder

#### ► Task 1: Create a new folder

1. Log on to the 10961B-LON-CL1 virtual machine as **Adatum\Administrator**.
2. To read the full Help for the **New-Item** command, run:

```
Help New-Item -ShowWindow
```

Notice the **-Name** and **-ItemType** parameters, and review the command examples.

3. To create a new folder, run:

```
New-Item -Path c:\ -Name ScriptOutput -ItemType Directory
```

**Results:** After completing this exercise, you will have created a new folder on the file system.

## Exercise 2: Create a New PSDrive

### ► Task 1: Create a new PSDrive

1. To read the full Help for the **New-PSDrive** command, run:

```
Help New-PSDrive -ShowWindow
```

Notice the **-Name**, **-Root**, and **-PSProvider** parameters. Review the command examples.

2. To create a new drive, run:

```
New-PSDrive -Name Output -Root C:\ScriptOutput -PSProvider FileSystem
```

**Results:** After completing this exercise, you will have created a new, temporary PSDrive.

## Exercise 3: Create a New Registry Key

### ► Task 1: Create the registry key

1. To view the Help for the **New-Item** command, run:

```
Help New-Item -ShowWindow
```

2. To create a new registry key, run:

```
New-Item -Path HKCU:\Software -Name Scripts
```

**Results:** After completing this exercise, you will have created a new registry key.

## Exercise 4: Create a Registry Setting

### ► Task 1: Create a new registry setting

1. To view the help for **New-ItemProperty**, run:

```
Help New-ItemProperty -ShowWindow
```

Notice the command parameters and review the command examples.

2. To create a new registry setting, run:

```
New-ItemProperty -Path HKLM:\Software\Microsoft\Windows\CurrentVersion\Run -Name  
"Windows PowerShell" -Value  
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
```

**Results:** After completing this exercise, you will have added a new program to the autorun list.

## Exercise 5: Modify a WS-Management Setting

### ► Task 1: Modify a WS-MAN setting

1. To read the Help for the **Set-Item** command, run:

```
Help Set-Item -ShowWindow
```

2. To modify a WS-Management setting, run:

```
Set-Item -Path WSMan:\localhost\Service\MaxConnections -Value 250
```

### ► Task 2: To prepare for the next module

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

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

**Results:** After completing this exercise, you will have modified the maximum number of concurrent connections for Windows PowerShell remoting.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 5: Formatting Output

## Lab: Formatting Output

### Exercise 1: Formatting Command Output

- Task 1: Create a formatted display of computer system information
1. Remember that you can pipe the output of a command to **Get-Member** to see a list of properties, or to **Format-List** \* to see a list of properties and their values. On the LON-CL1 virtual machine, using this command:

```
Get-CimInstance -Class Win32_ComputerSystem
```

2. Create a list display that includes the following:
  - Computer name
  - Description
  - Domain
  - Manufacturer
  - Model
  - Number of processors
  - Installed physical memory in gigabytes (GB)
3. To create the specified list, run:

```
Get-CimInstance -Class Win32_ComputerSystem |  
Format-List -Property PSComputerName,Description,Domain,Manufacturer,  
Model,NumberOfProcessors,@{n='TotalPhysicalMemory';e={$PSItem.TotalPhysicalMemory /  
1GB}}
```

► Task 2: Create a formatted table of process information

1. Display a table of running processes. The table must include the following:
  - Process name
  - ID
  - Virtual memory in megabytes (MB) to two decimal places
  - Physical memory in megabytes (MB) to two decimal places
2. The table should not have additional space between the columns. Redirect the table to a text file that is named **Procs.txt**.
3. To create the specified list, run:

```
Get-Process |  
Format-Table -Property Name,ID,@{n='VM(MB)';e={$PSItem.VM /  
1MB};formatString='N2'},@{n='PM(MB)';e={$PSItem.PM / 1MB};formatString='N2'} -  
AutoSize |  
Out-File Procs.txt
```

► **Task 3: Display a table of processes organized by base priority level**

1. Display tables of running processes. The tables must appear identical to the output normally produced by **Get-Process**. Using a single command, create one table for each value of the process' BasePriority property.
2. To display the specified list, run:

```
Get-Process | Sort BasePriority | Format-Table -GroupBy BasePriority
```

► **Task 4: Display a list of local TCP/IP routes**

1. Using the command **Get-NetRoute**, display a table of TCP/IP routes. The table must include the following:
  - Route address family
  - Route metric
  - Type of route
  - Destination prefix
2. Destination prefix must be right-aligned. The table must not include additional space between columns.
3. To create the specified table, run:

```
Get-NetRoute |  
Format-Table -Property AddressFamily,RouteMetric,TypeOfRoute,  
@{n='DestinationPrefix';e={$PSItem.DestinationPrefix};align='right'} -AutoSize
```

**Results:** After completing this exercise, you will have created various commands that produce formatted output.

## Exercise 2: Reproducing Specified Output

► **Task 1: Write a command that displays file names and sizes as specified**

1. Write a command that will display a list of all files having an .exe file name extension in the **C:\Windows** directory. Your output must look exactly as follows:

Name	Size(KB)
explorer.exe	2,273.76
HelpPane.exe	950.50
DfsrAdmin.exe	231.00
notepad.exe	212.50
regedit.exe	148.00
splwow64.exe	123.00
bfsvc.exe	55.50
hh.exe	17.00
winhlp32.exe	10.50
write.exe	10.50

2. To produce the specified output, run the command in E:\Mod05\Labfiles\Exercise2-Task1.ps1.

► **Task 2: Write a command that displays event log entries as specified**

1. Event log entries have a **TimeGenerated** and **TimeWritten**. The difference between those two times is typically zero. However, it is possible for there to be a delay between the time that the event is generated and the time that it is written into the log.
2. Display the most recent 20 entries from the Security event log. Calculate the difference between the time each event was generated and the time that it was written. Display the list exactly as shown here, with the largest time difference shown first, and the smallest time difference shown last.
3. You will use four commands to perform this task. Your output should be similar to this:

EventID	TimeDifference
4672	00:00:00
4634	00:00:00
4672	00:00:00
4624	00:00:00
4634	00:00:00
4624	00:00:00
4672	00:00:00
4648	00:00:00
4624	00:00:00
4672	00:00:00
4648	00:00:00
4624	00:00:00
{continues}	

4. To produce the specified output, run the command in: E:\Mod05\Labfiles\Exercise2-Task2.ps1.

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have written commands to reproduce specified output.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 6: Querying Management Information by Using WMI and CIM

# Lab: Working with WMI and CIM

### Exercise 1: Querying Information by Using WMI

► Task 1: Query IP addresses

1. To find a repository class that lists IP addresses being used by a computer, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*configuration*' |  
Sort Name
```

Notice the **Win32\_NetworkAdapterConfiguration** class.

2. To retrieve all instances of the class showing DHCP IP addresses, run:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration |  
Where DHCPEnabled -eq $True |  
Select IPAddress
```

Remember that you can run the first command and pipe its output to **Get-Member** to see what properties are available.

► Task 2: Query operating system version information

1. To find a repository class that lists operating system information, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*operating*' |  
Sort Name
```

Notice the **Win32\_OperatingSystem** class.

2. To display a list of properties for the class, run:

```
Get-WmiObject -Class Win32_OperatingSystem | Get-Member
```

3. Notice the **Version**, **ServicePackMajorVersion**, and **BuildNumber** properties.

4. To display the specified information, run:

```
Get-WmiObject -Class Win32_OperatingSystem |  
Select Version,ServicePackMajorVersion,BuildNumber
```

► Task 3: Query computer system hardware information

1. To find a repository class that displays computer system information, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*system*' |  
Sort Name
```

Notice the **Win32\_ComputerSystem** class.

2. To display a list of instance properties and values, run:

```
Get-WmiObject -Class Win32_ComputerSystem |
```

```
Format-List -Property *
```

Remember that **Get-Member** does not display property values, but **Format-List** can.

3. To display the specified information, run:

```
Get-WmiObject -class Win32_ComputerSystem |  
Select Manufacturer,Model,@{n='RAM';e={$PSItem.TotalPhysicalMemory}}
```

#### ► Task 4: Query service information

1. To find a repository class that contains information about services, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*service*' |  
Sort Name
```

Notice the **Win32\_Service** class.

2. To display a list of instance properties and values, run:

```
Get-WmiObject -Class Win32_Service |  
FL *
```

3. To display the specified information, run:

```
Get-WmiObject -Class Win32_Service -Filter "Name LIKE '%S'" |  
Select Name,State,StartName
```

**Results:** After completing this lab, you will have queried repository classes by using WMI commands.

## Exercise 2: Querying Information by Using CIM

### ► Task 1: Query user accounts

1. To find a repository class that lists user accounts, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*user*' |  
Sort Name
```

Notice the **Win32\_UserAccount** class.

2. To display a list of class properties, run:

```
Get-CimInstance -Class Win32_UserAccount |  
Get-Member
```

3. To display the specified information, run:

```
Get-CimInstance -Class Win32_UserAccount |  
Format-Table -Property Caption,Domain,SID,FullName,Name
```

You should now see a returned list of all domain and local accounts.

### ► Task 2: Query BIOS information

1. To find a repository class that contains BIOS information, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*bios*' |  
Sort Name
```

Notice the **Win32\_BIOS** class.

2. To display the specified information, run:

```
Get-CimInstance -Class Win32_BIOS
```

### ► Task 3: Query network adapter configuration information

1. To display a list of all local **Win32\_NetworkAdapterConfiguration** instances, run:

```
Get-CimInstance -Classname Win32_NetworkAdapterConfiguration
```

2. To display a list of all **Win32\_NetworkAdapterConfiguration** instances on LON-DC1, run:

```
Get-CimInstance -Classname Win32_NetworkAdapterConfiguration -ComputerName LON-DC1
```

### ► Task 4: Query user group information

1. To find a repository class that lists user groups, run:

```
Get-WmiObject -namespace root\cimv2 -list |  
Where Name -like '*group*' |  
Sort Name
```

Notice the **Win32\_Group** class.

2. To display the specified information, run:

```
Get-CimInstance -ClassName Win32_Group -ComputerName LON-DC1
```

**Results:** After completing this exercise, you will have queried repository classes by using CIM commands.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Invoking Methods

### ► Task 1: Invoke a CIM method

1. To restart LON-DC1, run:

```
Invoke-CimMethod -ClassName Win32_OperatingSystem -ComputerName LON-DC1 -MethodName Reboot
```

You should see a prompt in LON-CL1 console saying **ReturnValue=0** and **PSComputerName =LON-DC1**.

2. Switch to the LON-DC1 virtual machine and you should see it restarting.

### ► Task 2: Invoke a WMI method

1. Switch back to the LON-CL1 virtual machine
2. Right-click in the lower-left corner of the desktop, and then click **Computer Management** console.
3. Go to **Services and Applications**, and then select **Services**.
4. Locate the **Windows Remote Management** (WS-Management) service, and note the **Startup Type**. (The service is also known by the WinRM abbreviation, which is called out in the service description).
5. To change the start mode of the specified service, run:

```
Get-WmiObject -Class Win32_Service -Filter "Name='WinRM'" |  
Invoke-WmiMethod -Name ChangeStartMode -Argument 'Automatic'
```

6. Check the status of the WinRM service and verify it has changed.

### ► Task 3: To prepare for the next module

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

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

**Results:** After completing this exercise, you will have used CIM and WMI commands to invoke methods of repository objects.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 7: Preparing for Scripting

## Lab: Working with Security in Windows PowerShell

### Exercise 1: Configure Security

- ▶ Task 1: Review the execution policy
  - Run the following command to verify the current execution policy:

```
Get-ExecutionPolicy
```

The execution policy should be **Restricted**.

- ▶ **Task 2: Run a script**

1. To create the script file, run:

```
"Get-Service" | Out-File C:\Test.ps1
```

2. To run the newly created non-code-signed file, run:

```
C:\Test.ps1
```

Verify that you receive an error.

3. To set the execution policy, run:

```
Set-ExecutionPolicy RemoteSigned
```

4. Answer Yes to the prompt by typing **Y**.

5. Run sample code-signed file **E:\Mod07\Democode\Test-CodesignedFile.ps1**.

Verify that the file runs successfully.

- ▶ **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have configured and tested the execution policy on a computer.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 8: Moving from Command to Script to Module

## Lab A: Moving from Command to Script

### Exercise 1: Test the Command

- ▶ Task 1: Test the Command
- 1. In a Windows PowerShell console window, run **ise**. Make sure that the Windows PowerShell ISE window title bar says **Administrator**.
- 2. Run the following command to ensure that the local execution policy is correct:

```
Set-ExecutionPolicy RemoteSigned
```

- 3. Answer Yes to change the Execution Policy.
- 4. On the **File** menu, click **Open**.
- 5. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise1.ps1**.
- 6. Click **Run Script** in the Windows PowerShell ISE toolbar, or press F5.

**Results:** After completing this exercise, you will have tested the command and verified its functionality.

## Exercise 2: Parameterize Changing Values

### ► Task 1: Identify changing values in a command

1. On the **File** menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise1.ps1** if it is not already open.
3. Identify two values in this command that might have to change every time someone runs the command.
4. The **-Filter** parameter controls which kinds of logical disks are returned by the command. Local fixed drives are represented by the “**DriveType=3**” value. This could be changed to examine removable drives.
5. The **-ComputerName** parameter specifies the computer on which you want to run the CIM operation controls. The default value is **localhost**. This value could be changed to examine a different computer.

### ► Task 2: Create a new script file to work in

1. In the Windows PowerShell ISE, click the **New** button on the toolbar, or press Ctrl+N.
2. On the **File** menu, click **Save As** to save the empty file as a script.
3. Browse to drive C and save the file as **Get-DiskInfo.ps1**. If you cannot save the file, the Windows PowerShell ISE was not opened as Administrator.

### ► Task 3: Add a parameter block to the script

1. At the top of your blank Get-DiskInfo.ps1 script, add the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$ComputerName,
    [int]$DriveType = 3
)
```

This creates a new parameter block that includes the cmdlet binding attribute. The block defines two parameters and assigns a default value to the second parameter.

2. Press Ctrl+S to save the file.
3. Close the file.

### ► Task 4: Add parameters and use parameters in the command

**Note:** For the remaining labs in this module the Lab Answer Key will instruct you to open a file that contains all of the answers up to that point. The Lab Answer Key will then instruct you to make any necessary changes for that task. You should review the lab instructions to understand why you are making those changes.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise2-Task4.ps1**.
3. On the **File** menu, click **Open**.
4. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise1.ps1**.
5. In the Exercise1.ps1 script, highlight the lines 1 to 6 and press Ctrl+C.
6. Click the tab for the **Exercise2-Task4.ps1** script.

7. Point to the beginning of line 6.
8. Press Ctrl+V.
9. On the **File** menu, click **Save As**.
10. In the **File name** box, type **C:\Get-DiskInfo.ps1**, and then press Enter.
11. In the **Confirm Save As** dialog box, click **Yes**.
12. In the script, replace the value **3** with **\$DriveType**.
13. In the script, replace the value **localhost** with **\$ComputerName**.
14. Press Ctrl+S to save the file.
15. Close the file.

#### ► Task 5: Test the script

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise2-Task5.ps1**.
3. Press F5.
4. At the **ComputerName** prompt, type **LON-CL1**, and then press **Enter**.
5. Verify that the script finishes without error.

**Results:** After completing this exercise, you will have identified and parameterized changing values in the command.

## Exercise 3: Add verbose output

### ► Task 1: Add verbose output

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise3-Task1.ps1**.
3. On the **File** menu, click **Save As**.
4. In the **File name** box, type **C:\Get-DiskInfo.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, Click **Yes**.
6. On line 7 of the script, add the following command:

```
Write-Verbose "Getting drive types of $DriveType from $ComputerName"
```

7. Press Ctrl+S to save the file.

### ► Task 2: Test the script

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise3-Task2.ps1**.
3. On the **File** menu, click **Save As**.
4. In the **File name** box, type **C:\Get-DiskInfo.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press Ctrl+D.
7. At the console prompt, type the following, and then press Enter:

```
Cd \
```

And ensure that the console prompt is focused on the C:\ drive

8. At the console prompt, type the following, and then press Enter:

```
.\Get-DiskInfo -Comp localhost
```

9. At the console prompt, type the following, and then press Enter:

```
.\Get-DiskInfo -Comp localhost -Verbose
```

10. Press Ctrl+I to return to the Script Pane.

**Results:** After completing this exercise, you will have changed your script so that it produces verbose output.

## Exercise 4: Add Comment-Based Help

### ► Task 1: Add comment-based help

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Browse **to E:\Mod08\Labfiles\LabA\** and open **Exercise4-Task1.ps1**.
3. On the **File** menu, click **Save As**.
4. In the **File name** box, type **C:\Get-DiskInfo.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. On line 1 of the script, add the following:

```
<#
 .SYNOPSIS
Retrieves disk space information.
.DESCRIPTION
Retrieves disk information from a single computer.
.PARAMETER ComputerName
The name of the computer to query.
.PARAMETER DriveType
The type of drive to query. Defaults to 3, representing local fixed disks.
.EXAMPLE
.\Get-DiskInfo -ComputerName localhost -Verbose
#>
```

7. Press Ctrl+S to save the file.

### ► Task 2: Test the script

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the File menu, click **Open**.
2. Browse to **E:\Mod08\Labfiles\LabA\** and open **Exercise4-Task2.ps1**.
3. On the **File** menu, click **Save As**.
4. In the **File name** box, type **C:\Get-DiskInfo.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press Ctrl+D to switch to the Console pane.
7. At the console prompt, type the following, and then press Enter:

```
Cd \
```

Again, ensuring the focus in the console is on the C:\ drive

8. At the console prompt, type the following, and then press Enter:

```
Help .\Get-DiskInfo -ShowWindow
```

9. Close the C:\Get-DiskInfo.ps1 Help window.
10. Press Ctrl+I to return to the Script Pane.

**Results:** After completing this exercise, you will have added documentation to your script by using comment-based help.

## Lab B: Moving from Script to Function to Module

### Exercise 1: Convert the Script to a Function

#### ► Task 1: Add the function declaration

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE.
2. On the **File** menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise1-Task1.ps1**.
4. Scroll down to Line 36 and delete everything below that.
5. Press **Ctrl+A** to select all the content.
6. Press **Tab** to indent the content.
7. On line 1, add the following:

```
Function Get-DiskInfo {
```

8. On line 28, add the following:  

```
}
```
9. On the **File** menu, click **Save as**.
10. In the **File name** box, type **C:\Tools.ps1**, and then press Enter. If you are prompted in the **Confirm Save As** dialog box, click **Yes**.

#### ► Task 2: Execute the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise1-Task2.ps1**.
3. Scroll down to Line 36 and delete everything below that.
4. On line 30, add the following:

```
Get-DiskInfo -Comp localhost
```

5. On the **File** menu, click **Save as**.
6. In the **File name** box, type **C:\Tools.ps1**, and then press Enter.
7. In the **Confirm Save As** dialog box, click **Yes**.

► **Task 3: Test the script**

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, select **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise1-Task3.ps1**.
3. On the **File** menu, select **Save as**.
4. In the **File name** box, type **C:\Tools.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press F5.

**Results:** After completing this exercise, you will have converted the code in your script into a function.

## Exercise 2: Save the Script as a Script Module

### ► Task 1: Remove the function call

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise2-Task1.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type **C:\Tools.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Delete line 30.
7. Press Ctrl+S.

### ► Task 2: Save the script as a script module

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. On the taskbar, click the **File Explorer** icon to open a new **File Explorer** window.
2. In the left tree view, **Expand this PC**.
3. Expand the **Documents** library and click **Documents**.
4. On the ribbon, click **New folder**.
5. Type **WindowsPowerShell** and press Enter.
6. Double-click the **WindowsPowerShell** folder.
7. On the ribbon, click the **New folder** button.
8. Type **Modules** and press Enter.
9. Double-click the **Modules** folder.
10. On the ribbon, click **New folder**.
11. Type **MyTools** and press Enter.
12. Close the **File Explorer** window.
13. Click the **Windows PowerShell ISE** window.
14. On the **File** menu, click **Open**.
15. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise2-Task2.ps1**.
16. On the File menu, click **Save as**.
17. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.

### ► Task 3: Test the script module

**Note:** Ensure that you complete the prior task before you attempt this task.

1. In the Windows PowerShell ISE, press Ctrl+D.
2. Type the following and press Enter:

```
Get-DiskInfo -comp localhost
```

3. Type the following and press Enter (because the module may not be loaded, this command may produce an error that you may ignore):

```
Remove-Module MyTools
```

**Results:** After completing this exercise, you will have saved your script as a script module.

## Exercise 3: Add Debugging Breakpoints

### ► Task 1: Add a breakpoint

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise3.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTool s.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. On line 23, add the following:

```
Write-Debug "About to query $computername"
```

7. Press Ctrl+S to save the file.
8. Close the file.

### ► Task 2: Test the script module

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE and then on the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabB** and open **Exercise3-Task2.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTool s.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press Ctrl+D.
7. Type the following and press Enter:

```
Get-DiskInfo -Comp localhost
```

8. Type the following and press Enter:

```
Get-DiskInfo -Comp localhost -Debug
```

If you do not receive Confirm message dialog type the following and continue again from step 8

```
Import-Module MyTools
```

9. Press **S** to suspend the operation.
10. Type the following and press Enter:

```
$ComputerName
```

11. Type the following and press Enter:

```
Exit
```

12. Press **Y**.

13. Type the following and press Enter:

```
Remove-Module MyTools
```

**Results:** After completing this exercise, you will have added debugging breakpoints to the MyTools script module.

## Lab C: Implementing Basic Error Handling

### Exercise 1: Add Error Handling to a Function

- ▶ **Task 1: Add error handling to a function**

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Open the Windows PowerShell ISE as Administrator.
2. On the **File** menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabC\** and open **Exercise1-Task1.ps1**.
4. On the **File** menu, click **Save as**.
5. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
6. Click **Yes** in the **Confirm Save As** dialog box.
7. Use the mouse to highlight lines 25 through 29 in the script.
8. Press Tab.
9. On line 24, add the following:

```
Try {
```

10. On line 30, add the following:

```
} Catch {
    Write-Verbose "Error querying $ComputerName"
}
```

11. Modify line 25 to read as follows:

```
Get-WMIObject -Class Win32_LogicalDisk -Filter "DriveType=$DriveType" -ComputerName
$ComputerName -ErrorAction Stop |
```

12. Press Ctrl+S to save the file.

## ► Task 2: Test the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Open the Windows PowerShell ISE as Administrator.
2. On the **File** menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabC\** and open **Exercise1-Task2.ps1**.
4. On the File menu, click **Save as**.
5. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.ps1** and press Enter.
6. In the **Confirm Save As** dialog box, click **Yes**.
7. Press Ctrl+D to switch to the Console pane.
8. Type the following and press Enter (because the module may not be loaded, this command may produce an error that you can safely ignore):

```
Remove-Module MyTools
```

9. Type the following and press Enter:

```
Get-DiskInfo -Comp BAD -Verbose
```

You should receive verbose output as the script runs, ultimately resulting in an error message when querying BAD, because it doesn't exist.

Note: If you do not receive the verbose output type the following and try step 9 again

```
Import-Module MyTools
```

10. Type the following to remove the module, and then press Enter:

```
Remove-Module MyTools
```

**Results:** After completing this exercise, you will have added error handling to an existing function.

## Exercise 2: Add Error Handling to a New Function

### ► Task 1: Identify potential error points

1. Open the Windows PowerShell ISE as Administrator.
2. On the **File** menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabC** and open **Exercise2-Task1.ps1**.
4. Identify the **Get-EventLog** command as the command that could cause an error if an incorrect computer name was specified.
5. Press F5 to run the script
6. Notice the verbose output informing you it is connecting to "BAD" computer name and the resultant red error text saying it could not find the path specified. i.e. it could not find the computer.

### ► Task 2: Add error handling

**Note:** Ensure that you complete the previous task before you attempt this task.

1. On line 9 of Exercise2-Task1.ps1, add the following:

```
Try {
```

2. On line 12, add the following:

```
} Catch {
    Write-Verbose "Failed to connect to $ComputerName"
}
```

3. Modify line 10 to read as follows:

```
Get-EventLog -LogName Security -Newest $MaxEntries -ErrorAction Stop -ComputerName
$ComputerName |
```

4. Press Ctrl+S to save the file

### ► Task 3: Test the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Open the Windows PowerShell ISE.
2. On the **File** menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabC** and open **Exercise2-Task2.ps1**.
4. Press F5 to run the script

You should receive verbose output when connecting to BAD.

**Results:** After completing this exercise, you will have added error handling to a function that someone else wrote.

## Lab D: Creating an Advanced Function

## Exercise 1: Test an Existing Command

### ► Task 1: Test the command

1. Open the Windows PowerShell ISE as Administrator.
2. On the File menu, click **Open**.
3. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise1.ps1**.
4. Press F5 to run the script. It should run successfully and output details regards the operating system.

### ► Task 2: Identify changeable values

- In the Exercise1.ps1 script, identify **localhost** as a value that will have to change.

**Results:** After completing this exercise, you will have tested an existing command and identified changeable values.

## Exercise 2: Create a Parameterized Function

### ► Task 1: Create the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu of the Windows PowerShell ISE, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Start.ps1**.
3. On line 32, add the following:

```
function Get-OSInfo {  
}
```

4. On the **File** menu of the Windows PowerShell ISE, click **Open**.
5. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise1.ps1**.
6. In the **Exercise1.ps1** highlight lines 1 to 7 and click **Ctrl+C**.
7. Click the tab for the **Start.ps1** script.
8. Point above the last } in the file and press **Ctrl+V**.
9. Highlight the contents of the **Get-OSInfo** function and press **Tab**.
10. On the **File** menu, click **Save as**.
11. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.ps1**, and then press Enter.
12. In the **Confirm Save As** dialog box, click **Yes**.

### ► Task 2: Add a parameter block

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise2-Task2.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.ps1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. On line 33, add the following:

```
[CmdletBinding()]  
param(  
    [Parameter(Mandatory=$True)]  
    [String]$ComputerName  
)
```

7. Press **Ctrl+S**.

### ► Task 3: Parameterize the command

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise2-Task3.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTool s.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Modify line 38 to read as follows:

```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $ComputerName |
```

7. Press Ctrl+S.

### ► Task 4: Test the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise2-Task4.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTool s.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press Ctrl+D.
7. Type the following and press Enter (if this command produces an error, ignore it):

```
Remove-Module MyTools
```

8. Type the following and press Enter:

```
Get-OSInfo -comp localhost
```

Note: If you receive an error stating that the "The client cannot connect to the destination specified in the request....." please do the following

Open the Windows PowerShell Console as administrator and type the following command and press **Enter**

```
WinRM qc
```

When prompted to **Make these changes [y/n]?** type **Y** and then press **Enter**. Repeat when prompted.

9. Type the following and press Enter:

```
Remove-Module MyTools
```

**Results:** After completing this exercise, you will have created a parameterized function by using the provided command.

MCT UCE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Handle Multiple Targets

### ► Task 1: Modify a parameter

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise3-Task1.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Modify line 36 to read as follows:

```
[string[]]$ComputerName
```

7. Press Ctrl+S.

### ► Task 2: Implement an enumerating loop

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise3-Task2.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Use the pointer to highlight lines 39 through 43.
7. Press Tab.
8. On line 38, add the following:

```
ForEach ($name in $ComputerName) {
```

9. On line 44, add the following:

```
}
```

10. Modify line 39 to read as follows:

```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $name |
```

11. Press Ctrl+S.

### ► Task 3: Test the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise3-Task3.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press **Ctrl+D**.
7. Type the following and press Enter:

```
Get-OSInfo -comp localhost,LON-DC1
```

If you are receiving an error when running the above command, run the command from the Windows PowerShell console rather than the ISE. For example, do the following:

Open the Windows PowerShell as Administrator,

At the prompt type the following and press **Enter**

```
Cd C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools
```

Then repeat the command outlined earlier in step 7.

8. Type the following and press Enter:

```
Remove-Module MyTools
```

**Results:** After completing this exercise, you will have changed a function to accept multiple computer names as input.

## Exercise 4: Add error handling

### ► Task 1: Add error handling

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise4-Task1.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. On line 39, add the following:

```
Try {
```

7. On line 45, add the following:

```
} Catch {  
    Write "Error connecting to $name"  
}
```

8. Modify line 40 to read as follows:

```
Get-CimInstance -ClassName Win32_OperatingSystem -ErrorAction Stop -ComputerName  
$name |
```

9. Press Ctrl+S.

### ► Task 2: Test the function

**Note:** These instructions provide a starting point that does not require you to have completed any prior lab tasks. The starting point includes all tasks up to this one. If you are working on your own script, you can compare it to the starting point before you attempt this task.

1. Close all open files in the ISE. On the **File** menu, click **Open**.
2. Navigate to **E:\Mod08\Labfiles\LabD\** and open **Exercise4-Task2.ps1**.
3. On the **File** menu, click **Save as**.
4. In the **File name** box, type  
**C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools\MyTools.psm1**, and then press Enter.
5. In the **Confirm Save As** dialog box, click **Yes**.
6. Press Ctrl+D.
7. Type the following and press Enter:

```
Get-OSInfo -comp localhost,BAD,LON-CL1
```

If you are receiving an error when running the above command, run the command from the Windows PowerShell console rather than the ISE. For example, do the following:

Open the Windows PowerShell as Administrator,

At the prompt type the following and press **Enter**

```
Cd C:\Users\Administrator.ADATUM\Documents\WindowsPowerShell\Modules\MyTools
```

Then type the following and press Enter

```
Remove-Module MyTools
```

Then repeat the command outlined earlier in step 7.

8. Type the following and press Enter:

```
Remove-Module MyTools
```

► **Task 3: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have added error handling to a function.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 9: Administering Remote Computers

## Lab A: Using Basic Remoting

### Exercise 1: Enabling Remoting on the Local Computer

- ▶ Task 1: Enable remoting for incoming connections
- 1. Ensure you are signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd** with the Windows PowerShell console open.
- 2. To ensure you have the correct execution policy in place, run:

```
set-executionPolicy RemoteSigned
```

- 3. Answer **Yes** or **Y** to confirm the change.
- 4. On the **LON-CL1** computer, run **Enable-PSremoting**. Answer Yes to all prompts by entering **Y**. This will enable remoting.
- 5. To find a command that can list session configurations, run:

```
help *sessionconfiguration*
```

Notice the **Get-PSSessionConfiguration** command.

- 6. To list session configurations, run:
- ```
Get-PSSessionConfiguration
```
- 7. Verify that 2 to 4 session configurations were created.

**Results:** After completing this exercise, you will have enabled remoting on the client computer.

## Exercise 2: Performing One-to-One Remoting

► **Task 1: Connect to the remote computer and install an operating system feature on it.**

1. Ensure you are still signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. On **LON-CL1**, to establish a one-to-one connection to LON-DC1, run:

```
Enter-PSSession -ComputerName LON-DC1
```

3. After you are connected, to install the Network Load Balancing (NLB) feature on LON-DC1, run:

```
Install-WindowsFeature NLB
```

4. Wait for the command to be completed.
5. To disconnect, run:

```
Exit-PSSession
```

► **Task 2: Test multihop remoting**

1. To establish a one-to-one remoting connection to LON-DC1, run:

```
Enter-PSSession -ComputerName LON-DC1
```

2. To establish a connection from LON-DC1 to LON-CL1, run:

```
Enter-PSSession -ComputerName LON-CL1
```

You should receive an error that is indicative of the second hop. By default, you cannot establish a connection through an already-established connection.

3. To close the connection, run:

```
Exit-PSSession
```

► **Task 3: Observe remoting limitations**

1. Ensure you are signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.

2. To establish a one-to-one connection to LON-CL1, run:

```
Enter-PSSession -ComputerName localhost
```

3. Run:

```
Notepad
```

Notice that the shell seems to stop responding while it waits for Notepad to open, because Notepad is a graphical application and the shell has no way to display the graphical user interface (GUI).

4. Press **Ctrl+C** to cancel the process and return to a shell prompt.
5. To disconnect, run:

---

**Exit-PSSession**

**Results:** After completing this exercise, you will have connected to a remote computer and performed maintenance tasks on it.

## Exercise 3: Performing One-to-Many Remoting

► **Task 1: Retrieve a list of physical network adapters from two computers.**

1. Ensure you are still signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. On **LON-CL1**, to find a command that can list network adapters, run:

```
help *adapter*
```

Notice the **Get-NetAdapter** command.

3. To view the Help for the command, run:

```
help Get-NetAdapter
```

Notice the **-Physical** parameter.

4. To run the command on LON-DC1 and LON-CL1 by means of remoting, run:

```
Invoke-Command -ComputerName LON-CL1,LON-DC1 -ScriptBlock { Get-NetAdapter -Physical }
```

► **Task 2: Compare the output of a local command to that of a remote command.**

1. To view the members of a **Process** object, run:

```
Get-Process | Get-Member
```

2. To view the members from a remote **Process** object, run:

```
Invoke-Command -ComputerName LON-DC1 -ScriptBlock { Get-Process } | Get-Member
```

Notice that the second set of results does *not* include **MemberType** of **Method**, and that the **TypeName** is different. That is the remote value is deserialized in comparison to the local output.

**Results:** After completing this exercise, you will have run commands against multiple remote computers.

## Lab B: Using Remoting Sessions

### Exercise 1: Using Implicit Remoting

► **Task 1: Create a persistent remoting connection to a server**

1. On **10961B-LON-CL1**, logged in as **Adatum\Administrator** with password **Pa\$\$w0rd**, in the Windows PowerShell console, create a persistent connection to LON-DC1 and store it in a variable. Run:

```
$dc = New-PSSession -ComputerName LON-DC1
```

2. To view the session list in the variable, run:

```
$dc
```

Verify that the connection is available.

► **Task 2: Import and use a module from a server**

1. To display a list of modules on LON-DC1, run:

```
Get-Module -ListAvailable -PSSession $dc
```

2. To find a module on LON-DC1 that can work with Server Message Blocks (SMB) shares, run:

```
Get-Module -ListAvailable -PSSession $dc |  
Where { $_.Name -Like '*share*' }
```

3. To import the module from LON-DC1 to your local computer, and to add the prefix **DC** to the important commands' nouns, run:

```
Import-Module -PSSession $dc -Name SMBShare -Prefix DC
```

4. To display a list of shares on LON-DC1, run:

```
Get-DCSMBShare
```

Because this command implicitly runs on LON-DC1, the command will display shares for that computer.

5. To display a list of shares on the local computer, run:

```
Get-SMBShare
```

Because you added the **DC** prefix to the imported commands, the local ones are still available by their original name.

► **Task 3: Close all open remoting connections**

1. Run:

```
Get-PSSession | Remove-PSSession
```

2. While not explicitly called out in the sample answer script provided, E:\Mod09\Labfiles\ImplicitRemoting.ps1. To verify the remoting connection has been closed you can run the following:

```
Get-PSSession
```

Verify no sessions are returned.

**Results:** After completing this exercise, you will have used implicit remoting to import and run commands from a remote computer.

## Exercise 2: Multicomputer Management

### ► Task 1: Create remoting sessions to two computers

1. Ensure you're still signed in to **10961B-LON-CL1** as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. To create remoting sessions to LON-CL1 and LON-DC1, and to save those in a variable, run:

```
$computers = New-PSSession -ComputerName LON-CL1,LON-DC1
```

3. To verify the connections, run:

```
$computers
```

Verify that two connections are shown as available.

### ► Task 2: Create a report that displays Windows Firewall rules from two computers

1. To find a module capable of working with network security, run:

```
Get-Module -ListAvailable | Where { $_.Name -like '*security*' }
```

Notice the NetSecurity module.

2. To load the module into memory on LON-CL1 and LON-DC1, run:

```
Invoke-Command -Session $computers -ScriptBlock { Import-Module NetSecurity }
```

3. To find a command that can display Windows Firewall rules, run:

```
Get-Command -Module NetSecurity
```

Notice the **Get-NetFirewallRule** command. If you would like to review the Help for the command, run:

```
Help Get-NetFirewallRule -ShowWindow
```

Note: If the help isn't displaying correctly run the commands from steps 1 to 3 in the Windows PowerShell console as administrator rather than the Windows PowerShell ISE

4. To display a list of enabled firewall rules on LON-DC1 and LON-CL1, run:

```
Invoke-Command -Session $computers -ScriptBlock { Get-NetFirewallRule -Enabled True }  
|  
Select Name,PSComputerName
```

5. To unload the module on LON-DC1 and LON-CL1, run:

```
Invoke-Command -Session $computers -ScriptBlock { Remove-Module NetSecurity }
```

### ► Task 3: Create and display an HTML report that displays local disk information from two computers. Your report must include each computer's name, each drive's letter, and each drive's free space and total size in bytes.

1. To display a list of local hard drives, filtered to include only those with a drive type of 3, run:

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3"
```

2. To run the same command on LON-DC1 and LON-CL1 by means of remoting, run:

```
Invoke-Command -Session $computers -ScriptBlock { Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" }
```

3. To produce an HTML report containing the results of the previous command, run:

```
Invoke-Command -Session $computers -ScriptBlock { Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" } | ConvertTo-HTML -Property PSComputerName,DeviceID,FreeSpace,Size
```

► **Task 4: Close and remove all open remoting sessions**

- To remove all open remoting sessions, run:

```
Get-PSSession | Remove-PSSession
```

► **Task 5: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have performed several management tasks against multiple computers.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 10: Putting it All Together

## Lab: Provisioning a New Server Core Instance

### Exercise 1: Create a Parameterized Script

- Task 1: In the Script Pane, create a script named Set-ServerCoreInstance.ps1 that uses cmdlet binding and has a parameter block

1. Ensure you are signed into the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. Open File Explorer and create a folder named **C:\Scripts**. You will use this folder during the lab.
3. Open the Windows PowerShell ISE.
4. In the Console pane of the Integrated Scripting Environment (ISE), run:

```
Set-ExecutionPolicy RemoteSigned
```

Ensure that the execution policy is set to **RemoteSigned**.

5. Click **Yes** in the Execution Policy Change dialog box.
6. In the Script pane, add the attribute and parameter block as shown in E:\Mod10\Labfiles\Exercise\_01\_A.ps1:

```
[CmdletBinding()]
Param(
)
```

7. Press Ctrl+S and save the script as **C:\Scripts\Set-ServerCoreInstance.ps1**.

► Task 2: Add input parameters to the script. Perform this task in the Script Pane

1. Create the parameters as shown in E:\Mod10\Labfiles\Exercise\_01\_B.ps1. Your script should contain the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$MACAddress,
    $LocalCredential = (Get-Credential -Message "Provide credential for target
machine"),
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add
machine to domain"),
    [Parameter(Mandatory=$True)]
    [string]$NewComputerName,
    [Parameter(Mandatory=$True)]
    [string]$NewIPAddress,
    [Parameter(Mandatory=$True)]
    [string]$Role,
    [string]$Domain = "ADATUM",
    [Parameter(Mandatory=$True)]
    [string]$ScopeID,
    [Parameter(Mandatory=$True)]
    [string]$DHCPServerName
)
```

**Results:** After completing this exercise, you will have created the beginnings of a parameterized script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 2: Get the Dynamic IP Address of the New Server Core Computer

### ► Task 1: Discover the MAC address of the LON-SVR1 computer

**Note:** In a real environment, you would have access to the MAC address either on a label (for a physical computer) or through hypervisor configuration tools (for a virtual machine). In this task, you will discover the information manually.

1. On the 10961B-LON-SVR1 virtual machine, log on by using the user name **Administrator** and the password **Pa\$\$w0rd**.
2. Run:

```
ipconfig /all
```

Notice the 12-character physical address that is displayed as “**00-1A-2B-3C-4D-5E**”. Your MAC Address will be different. You will need this physical (MAC) address information throughout this lab. Ensure that you write it down clearly, in an easy-to-access location.

3. Run **Logoff** to log off from the virtual machine.
4. In the Console pane on LON-CL1, run:

```
$MACAddress = "01-23-45-01-23-45"
```

Replace “**01-23-45-01-23-45**” with the MAC address of the Server Core instance LON-SVR1. You can confirm that the variable was defined as you expected by typing **\$MACAddress** and pressing enter. The value that the variable contains will display and you can verify it. This could be done to double check all the variables that you will define as you proceed through this lab.

**Note:** You will not log on to LON-SVR1 again.

### ► Task 2: In the Console pane, look up the IP address of the Server Core instance in the DHCP server

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. In the Windows PowerShell ISE, display the Console pane by clicking toolbar buttons or the blue **Hide/Show arrow** icon. Or, press **Ctrl+D**.
3. To find a command that can query DHCP leases, run:

```
Help *lease*
```

Notice the command **Get-DhcpServerv4Lease**. Read the command Help, and notice that it requires you to specify a DHCP scope.

4. To find a command that can query DHCP scopes, run:

```
help *scope*
```

Notice the command **Get-DhcpServerv4Scope**.

5. To save the computer name in a variable, run:

```
$DHCPHostName = "LON-DC1"
```

6. In the Console pane, display a list of scopes by running:

```
Get-DhcpServerv4Scope -ComputerName $DHCPHostName
```

Write down the IP address of the only scope.

7. To save the scope ID in a variable, run:

```
$ScopeID = "10.0.0.0"
```

8. To display a list of all leases in that scope, run:

```
Get-DHCPServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName
```

9. To store the IP address of the Server Core instance, run:

```
$OldIPAddress = Get-DHCPServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName  
| Where-Object { $PSItem.ClientID -eq $MACAddress } |  
Select-Object -ExpandProperty IPAddress |  
Select-Object -ExpandProperty IPAddressToString
```

10. To convert the IP address to a string object, run:

```
$OldIPAddress = "$OldIPAddress"
```

11. To save a computer name to a variable, run:

```
$NewComputerName = "LON-SVR2"
```

**Note:** The existing name of the target computer is LON-SVR1. We will rename it to the name defined in the variable \$NewComputername as part of this process i.e. LON-SVR2.

### ► Task 3: Update your script to look up the IP address

1. Modify your script as shown in E:\Mod10\Labfiles\Exercise\_02\_B.ps1. Your script should contain the following:

```
[CmdletBinding()  
Param(  
    [Parameter(Mandatory=$True)]  
    [string]$MACAddress,  
    $LocalCredential = (Get-Credential -Message "Provide credential for target  
machine"),  
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add  
machine to domain"),  
    [Parameter(Mandatory=$True)]  
    [string]$NewComputerName,  
    [Parameter(Mandatory=$True)]  
    [string]$NewIPAddress,  
    [Parameter(Mandatory=$True)]  
    [string]$Role,  
    [string]$Domain = "ADATUM",  
    [Parameter(Mandatory=$True)]  
    [string]$ScopeID,  
    [Parameter(Mandatory=$True)]  
    [string]$DHCPServerName  
)  
$OldIPAddress = Get-DhcpServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName  
|  
    Where-Object { $PSItem.ClientId -eq $MACAddress } |  
    Select-Object -ExpandProperty IPAddress |  
    Select-Object -ExpandProperty IPAddressToString  
$OldIPAddress = "$OldIPAddress"
```

**Results:** After completing this exercise, you will have written a command to retrieve an IP address from the DHCP server based on the MAC address of the computer.

## Exercise 3: Create a DHCP Reservation for the Server Core Instance

### ► Task 1: In the Console pane, create the DHCP reservation

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. To find a command that can create a new DHCP reservation, run:

```
help *reservation*
```

Notice the command **Add-DhcpServerv4Reservation**.

3. In the variable **\$NewIPAddress**, store the string value **10.0.0.10**.

```
$NewIPAddress = "10.0.0.10"
```

4. Run:

```
Add-DhcpServerv4Reservation -ClientId $MACAddress -IPAddress $NewIPAddress -ScopeId  
$ScopeID -ComputerName $DHCPServerName  
End of RLO
```

**Note:** You can verify the reservation has been created by going to the **Start** page , typing **administrative** and then selecting **Administrative Tools**.In Administrative Tools, double-click **DHCP**.

The DHCP management console opens, and you should then right click **DHCP** and go to **Add Server...** to add **LON-DC1**. Once LON-DC1 is added look at the values under **lon-dc1\IPv4\Scope [10.0.0.0] 10.0.0.0/24\Reservations** ensuring **10.0.0.10** is listed.

### ► Task 2: In the Script Pane, update your script to create the DHCP reservation

1. Modify your script as shown in E:\Mod10\Labfiles\Exercise\_03\_A.ps1. Your script should contain the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$MACAddress,
    $LocalCredential = (Get-Credential -Message "Provide credential for target
machine"),
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add
machine to domain"),
    [Parameter(Mandatory=$True)]
    [string]$NewComputerName,
    [Parameter(Mandatory=$True)]
    [string]$NewIPAddress,
    [Parameter(Mandatory=$True)]
    [string]$Role,
    [string]$Domain = "ADATUM",
    [Parameter(Mandatory=$True)]
    [string]$ScopeID,
    [Parameter(Mandatory=$True)]
    [string]$DHCPServerName
)
$OldIPAddress = Get-DhcpServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName
|
Where-Object { $PSItem.ClientId -eq $MACAddress } |
Select-Object -ExpandProperty IPAddress |
Select-Object -ExpandProperty IPAddressToString
$OldIPAddress = "$OldIPAddress"
# Add a reservation
Add-DhcpServerv4Reservation -ClientId $MACAddress `
```

-IPAddress \$NewIPAddress -ScopeId \$ScopeID `

-ComputerName \$DHCPServerName

2. Press Ctrl+S.

**Results:** After completing this exercise, you will have created a DHCP reservation for the LON-SVR1 computer.

MCT UCT ONLY. STUDENT USE PROHIBITED

## Exercise 4: Modify the local TrustedHosts list

### ► Task 1: Save the TrustedHosts list to a variable. Perform this task in the Console pane

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. To save the existing TrustedHosts list, run:

```
$OriginalTrustedHosts = Get-Item WSMAN:\localhost\Client\TrustedHosts | Select-Object -ExpandProperty Value
```

3. If the Start Service dialog appears click **Yes** in the Start WinRM Service dialog box.
4. To verify the results, run:

```
$OriginalTrustedHosts
```

This may display a blank value if that is what was originally in your TrustedHosts list. Run **Dir WsMan:\localhost\Client** and note the value that is listed for the Name **TrustedHosts**. Verify that the value listed matches the TrustedHosts value that you just assigned to the **\$OriginalTrustedHosts** variable i.e. both may be blank.

### ► Task 2: In the Console pane, add the Server Core computer's IP address to the TrustedHosts list

1. To set the TrustedHosts list, run:

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value "$OldIPAddress"
```

2. Click **Yes** to confirm the WinRM Security Configuration dialog box.
3. To verify the list, run:

```
Dir WSMAN:\localhost\Client
```

Verify that the TrustedHosts item is now set to the same value contained in **\$OldIPAddress**.

4. To restore the list, run:

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value "$OriginalTrustedHosts"
```

Note the quotation marks around the **\$OriginalTrustedHosts** variable in the command above. This is to ensure the value is understood by the shell to be a string.

5. In the WinRM Security Configuration dialog box, click **Yes**.
6. To verify the list, run:

```
Dir WSMAN:\localhost\Client
```

Verify that TrustedHosts no longer contains the value in **\$OldIPAddress**.

7. Reset TrustedHosts to **\$OldIPAddress**.

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value $OldIPAddress
```

8. In the WinRM Security Configuration dialog box, click **Yes**.
9. Verify the TrustedHosts value again by typing

Dir WSMAN:\localhost\Client

10. Step 9 should display the TrustedHosts value as the \$OLDIPAddress variable, i.e. the IP Address of the remote virtual machine LON-SVR1 that we'll be connecting to as part of this lab. If this value is not present after step 9 you'll need to set it as such by repeating the steps 7 to 9 above in this task. This can also be done by using the subsequent scripts provided as we continue through the lab exercises.

► **Task 3: In the Script Pane, update your script to modify TrustedHosts**

1. Modify your script as shown in E:\Mod10\Labfiles\Exercise\_04\_B.ps1. Your script should contain the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$MACAddress,
    $LocalCredential = (Get-Credential -Message "Provide credential for target
machine"),
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add
machine to domain"),
    [Parameter(Mandatory=$True)]
    [string]$NewComputerName,
    [Parameter(Mandatory=$True)]
    [string]$NewIPAddress,
    [Parameter(Mandatory=$True)]
    [string]$Role,
    [string]$Domain = "ADATUM",
    [Parameter(Mandatory=$True)]
    [string]$ScopeID,
    [Parameter(Mandatory=$True)]
    [string]$DHCPServerName
)
$OldIPAddress = Get-DhcpServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName
|
Where-Object { $PSItem.ClientId -eq $MACAddress } |
Select-Object -ExpandProperty IPAddress |
Select-Object -ExpandProperty IPAddressToString
$OldIPAddress = "$OldIPAddress"
# Add a reservation
Add-DhcpServerv4Reservation -ClientId $MACAddress `

-IPAddress $NewIPAddress -ScopeId $ScopeID `

-ComputerName $DHCPServerName
# Save TrustedHosts
$OriginalTrustedHosts = Get-Item WSMAN:\localhost\Client\TrustedHosts | select -
ExpandProperty Value
# Set TrustedHosts
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value $OldIPAddress
# Restore TrustedHosts
Set-Item
WSMAN:\localhost\Client\TrustedHosts -Value "$OriginalTrustedHosts"
```

**Results:** After completing this exercise, you will have saved your TrustedHosts list, and added the Server Core computer's IP address to it.

## Exercise 5: Add a Role to the Server Core Instance

### ► Task 1: In the Console pane, add a role to the Server Core computer

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. To find a command that can add a Windows Feature, run:

```
help *feature*
```

Notice the **Add-WindowsFeature** or **Install-WindowsFeature** commands (they are the same; the first is an alias to the second).

3. To create the credential object, run:

```
$LocalCredential = Get-Credential
```

When prompted, provide the user name **Administrator** and the password **Pa\$\$w0rd**.

4. To add the Telnet Client role to the Server Core instance, run:

```
Invoke-Command -ComputerName $OldIPAddress -Credential $LocalCredential -ScriptBlock
{ Install-WindowsFeature Telnet-Client }
```

5. To check the windows feature installed correctly you could also sign in to LON-SVR1 and type **PowerShell** and press Enter to access the PowerShell prompt then at the prompt. The type **Get-WindowsFeature**. Scroll through the output until you come across **Telnet-Client**. Verify an "X" is present in the box beside it to indicate the feature is installed on the system.

### ► Task 2: In the Script Pane, update your script to add a role

1. Modify your script as shown in E:\Mod10\Labfiles\Exercise\_05\_A.ps1.
2. Your script should contain the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$MACAddress,
    $LocalCredential = (Get-Credential -Message "Provide credential for target
machine"),
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add
machine to domain"),
    [Parameter(Mandatory=$True)]
    [string]$NewComputerName,
    [Parameter(Mandatory=$True)]
    [string]$NewIPAddress,
    [Parameter(Mandatory=$True)]
    [string]$Role,
    [string]$Domain = "ADATUM",
    [Parameter(Mandatory=$True)]
    [string]$ScopeID,
    [Parameter(Mandatory=$True)]
    [string]$DHCPServerName
)
$OldIPAddress = Get-DhcpServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName
|
Where-Object { $PSItem.ClientId -eq $MACAddress } |
Select-Object -ExpandProperty IPAddress |
Select-Object -ExpandProperty IPAddressToString
$OldIPAddress = "$OldIPAddress"
# Add a reservation
Add-DhcpServerv4Reservation -ClientId $MACAddress `
```

-

```
-IPAddress $NewIPAddress -ScopeId $ScopeID `
```

```
# Save TrustedHosts
$OriginalTrustedHosts = Get-Item WSMAN:\localhost\Client\TrustedHosts |
    select -ExpandProperty value
# Set TrustedHosts
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value $OldIPAddress
# Install role
Invoke-Command -ComputerName $OldIPAddress ` 
    -Credential $LocalCredential ` 
    -ScriptBlock { Install-WindowsFeature Telnet-Client }
# Restore TrustedHosts
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value "$OriginalTrustedHosts"
```

**Results:** After completing this exercise, you will have added a role to the Server Core computer.

## Exercise 6: Add the Server Core Instance to the Domain

### ► Task 1: In the Console pane, add the computer to the domain

1. Ensure you are still logged on to the **10961B-LON-CL1** virtual machine as **Adatum\Administrator** with password **Pa\$\$w0rd**.
2. To find a command that will add a computer to a domain, run

```
help *computer*
```

Notice the **Add-Computer** command. Write down four parameters that are needed to add a computer to a domain, to rename the computer while adding it, and to restart the computer.

3. To save a domain credential in a variable, run:

```
$DomainCredential = Get-Credential
```

When prompted, provide the user name **ADATUM\Administrator** and the password **Pa\$\$w0rd**.

4. To add the computer to the domain, run:

```
Invoke-Command -ComputerName $OldIPAddress -Credential $LocalCredential -ScriptBlock
{ param($x,$y) Add-Computer -DomainName ADATUM -NewName $x -Credential $y -Restart }
-ArgumentList $NewComputerName,$DomainCredential
```



**Note:** It is expected that the target computer will restart and you will lose your connection. The LON-CL1 is attempting to connect to the old IP Address and will not successfully reconnect as the IP Address has been changed to the new address. In the console prompt you will see an error message saying “..Connection Lost..” attempting to reconnect.

If you are attempting to run the script and are unsuccessful it may be because changes have been made to the LON-SVR1 virtual machine outside of what was the original configuration, you may need to validate that the IPAddress, DHCP configuration etc is as expected and other variables listed in the command. A solution may be to revert the LON-DC1 and LON-SVR1 virtual machines to the StartImage snapshots virtual machines. There should be no need to revert the LON-CL1 virtual machine.

### ► Task 2: In the Script Pane, update your script to add and rename the computer

1. Modify your script as shown in E:\Mod10\Labfiles\Exercise\_06\_A.ps1. Your script should contain the following:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$True)]
    [string]$MACAddress,
    $LocalCredential = (Get-Credential -Message "Provide credential for target
machine"),
    $DomainCredential = (Get-Credential -Message "Provide domain credential to add
machine to domain"),
    [Parameter(Mandatory=$True)]
    [string]$NewComputerName,
    [Parameter(Mandatory=$True)]
    [string]$NewIPAddress,
    [Parameter(Mandatory=$True)]
    [string]$Role,
    [string]$Domain = "ADATUM",
    [Parameter(Mandatory=$True)]
    [string]$ScopeID,
    [Parameter(Mandatory=$True)]
```

```
[string]$DHCPServerName
)
$OldIPAddress = Get-DhcpServerv4Lease -ScopeId $ScopeID -ComputerName $DHCPServerName
|
Where-Object { $PSItem.ClientId -eq $MACAddress } |
Select-Object -ExpandProperty IPAddress |
Select-Object -ExpandProperty IPAddressToString
$OldIPAddress = "$OldIPAddress"
# Add a reservation
Add-DhcpServerv4Reservation -ClientId $MACAddress ` 
-IPAddress $NewIPAddress -ScopeId $ScopeID ` 
-ComputerName $DHCPServerName
# Save TrustedHosts
$OriginalTrustedHosts = Get-Item WSMAN:\localhost\Client\TrustedHosts | 
select -ExpandProperty value
# Set TrustedHosts
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value $OldIPAddress
# Install role
Invoke-Command -ComputerName $OldIPAddress ` 
-Credential $LocalCredential ` 
-ScriptBlock { Install-WindowsFeature Telnet-Client }
# Add to domain and rename
Invoke-Command -ComputerName $OldIPAddress ` 
-Credential $LocalCredential ` 
-ScriptBlock { param($x,$y) Add-Computer -DomainName ADATUM ` 
-NewName $x ` 
-Credential $y ` 
-Restart } ` 
-ArgumentList $NewComputerName,$DomainCredential
# Restore TrustedHosts
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value "$OriginalTrustedHosts"
```

2. Press Ctrl+S.

**Results:** After completing this exercise, you will have renamed the Server Core computer and added it to the domain.

## Exercise 7: Test the Completed Script

### ► Task 1: Run the completed script in the Console pane

1. Follow the instructions in E:\General\Revert.txt to revert the two virtual machines, **10961B-LON-DC1** and the **10961B-LON-SVR1**. You may also need to restart 10961B-LON-DC1 and 10961B-LON-SVR1, and then reset the execution policy to remote signed on 10961B-LON-CL1 after reverting them.
2. In the Console pane, run:

```
C:\Scripts\Set-ServerCoreInstance -NewComputerName LON-SVR2 -NewIPAddress 10.0.0.10 -Role Telnet-Client -MACAddress 00-15-5D-24-3D-14 -ScopeID 10.0.0.0 -DHCPServerName LON-DC1
```

In the preceding command, provide the MAC address of the Server Core instance in place of **00-15-5D-24-3D-14**. You should have written down the MAC address in an earlier exercise.

In the preceding command, provide the DHCP scope IP address in place of **10.0.0.0**. You should have written down the scope IP address in an earlier exercise.

When you are prompted for the local credential, provide the user name **Administrator** and password **Pa\$\$w0rd**.

When you are prompted for the domain credential, provide user name **ADATUM\Administrator** and password **Pa\$\$w0rd**.

3. Wait for the script to complete, and then wait a few minutes for the Server Core instance to restart.

**Note:** It is expected that the target computer will restart and you will lose your connection. The LON-CL1 is attempting to connect to the old IP Address and will not successfully reconnect as the IP Address has been changed to the new address. In the console prompt you will see an error message saying "...Connection Lost..." attempting to reconnect. You will then receive a WinRM Security Configuration prompt, and press Y to confirm after which the script will stop running.

If the script is still not running and you have reverted your Virtual machines or applied and snapshot and restarted you may want to check the variables for MACAddress, OldIPAddress on the LON-SVR1 virtual machine to ensure they are still as you established and noted earlier. Applying snapshots or having altered boot sequences after such could potentially affect some of the variables.

4. To verify that the computer is in the domain, on LON-CL1 in with Windows PowerShell console run:

```
Get-ADComputer -filter *
```

Confirm that LON-SVR2 is listed.

5. To verify the role installation in the Windows PowerShell ISE, run:

```
Get-WindowsFeature -ComputerName LON-SVR2
```

Confirm that the Telnet-Client role is listed as Installed.

6. To verify the TrustedHosts list, run:

```
Dir WSMAN:\localhost\Client
```

Confirm that the TrustedHosts item is empty.

7. This ensures that the TrustedHosts value has been returned to its original state, which is most likely empty, after been changed to the \$OLDIPAddress variable i.e. the original IP address of the LON-SVR1 machine. If the IP address were left in the TrustedHosts item this could be a security vulnerability.

► **Task 2: To prepare for the next module**

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

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

**Results:** After completing this exercise, you will have automated the provisioning process and verified the results.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 11: Using Background Jobs and Scheduled Jobs

## Lab A: Using Background Jobs

### Exercise 1: Starting Jobs

- ▶ Task 1: Start a Remoting Job
1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
  2. Type **Windows PowerShell** on the start screen, right click the **Windows PowerShell** tile, and click **Run as administrator**.
  3. To enable remoting on LON-CL1, run the following command and answer Yes by typing **Y** at each prompt:

```
Enable-PSRemoting
```

4. To start a remoting job that retrieves a list of physical network adapters from LON-DC1 and LON-CL1, run:

```
Invoke-Command -ScriptBlock { Get-NetAdapter -Physical } -ComputerName LON-DC1,LON-CL1 -AsJob -JobName RemoteNetAdapt
```

5. To start a remoting job that retrieves a list of Server Message Block (SMB) shares from LON-DC1 and LON-CL1, run:

```
Invoke-Command -ScriptBlock { Get-SMBShare } -ComputerName LON-DC1,LON-CL1 -AsJob -JobName RemoteShares
```

6. To start a remoting job that retrieves all instances of the **Win32\_Volume** class from every computer in Active Directory® Domain Services, run:

```
Invoke-Command -ScriptBlock { Get-CimInstance -ClassName Win32_Volume } -ComputerName (Get-ADComputer -Filter * | Select -Expand Name) -AsJob -JobName RemoteDisks
```

Because not every computer in the domain may be online, this job may not complete successfully. That is expected.

#### ▶ Task 2: Start a Local Job

1. To start a local job that retrieves all entries from the Security event log, run:

```
Start-Job -ScriptBlock { Get-EventLog -LogName Security } -Name LocalSecurity
```

2. To start a local job that produces 100 directory listings, run:

```
Start-Job -ScriptBlock { 1..100 | ForEach-Object { Dir C:\ -Recurse } } -Name LocalDir
```

This job is expected to take a very long time to complete. Do not wait for it to complete—proceed to the next task.

**Results:** After completing this exercise, you will have started jobs using two of the basic job types.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 2: Managing Jobs

### ► Task 1: Review Job Status

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. To display a list of running jobs, run:

```
Get-Job
```

3. To display a list of running jobs whose names start with **remote**, run:

```
Get-Job -Name Remote*
```

### ► Task 2: Stop a Job

1. To stop the **LocalDir** job, run:

```
Stop-Job -Name LocalDir
```

### ► Task 3: Retrieve Job Results

1. Run **Get-Job** until no jobs have a status of **Running**.
2. To receive the results of the **RemoteNetAdapt** job, run:

```
Receive-Job -Name RemoteNetAdapt
```

3. To receive the LON-DC1 results of the **RemoteDisks** job, run:

```
Get-Job -Name RemoteDisks |  
Select -Expand ChildJobs |
```

4. Where Location -eq 'LON-DC1' | Receive-Job

**Results:** After completing this exercise, you will have managed the jobs that you created in the previous exercise.

## Lab B: Using Scheduled Jobs

### Exercise 1: Creating a Scheduled Job

#### ► Task 1: Create Job Options

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. To create a new job option, run:

```
$option = New-ScheduledJobOption -WakeToRun -RunElevated
```

#### ► Task 2: Create a job trigger

1. To create a new job trigger, run:

```
$trigger1 = New-JobTrigger -Once -At (Get-Date).AddMinutes(10)
```

2. To create a second job trigger, run:

```
$trigger2 = New-JobTrigger -AtLogOn
```

#### ► Task 3: Create a Scheduled Job

1. To register the job, run:

```
Register-ScheduledJob -ScheduledJobOption $option  
-Trigger $trigger1  
-ScriptBlock { Get-EventLog -LogName Security }  
-MaxResultCount 5  
-Name LocalSecurityLog
```

2. To register a second job, run:

```
Register-ScheduledJob -ScheduledJobOption $option  
-Trigger $trigger2  
-ScriptBlock { Get-Process }  
-Name ProcList
```

3. To display a list of job triggers, run:

```
Get-ScheduledJob -Name LocalSecurityLog |  
Select -Expand JobTriggers
```

Write down the time shown.

4. Run:

```
Logoff
```

5. Wait until the time from step 3 has passed.
6. Log on to LON-CL1 using user name **ADATUM\Administrator** and the password **Pa\$\$w0rd**.

#### ► Task 4: Retrieve Job Results

1. Type **powershe**.
2. Right-click the Windows PowerShell icon and click **Run as administrator**.
3. To display a list of jobs, run:

```
Import-Module PSScheduledJob  
Get-Job
```

4. To receive the job results, run:

```
Receive-Job -Name LocalSecurityLog
```

5. To receive the job results, run:

```
Receive-Job -Name ProcList
```

#### ► Task 5: To prepare for the next module

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

1. On the host computer, start Hyper-V® Manager.
2. In the **Virtual Machines** list, right-click **10961B-LON-DC1**, and then click **Revert**.

3. In the **Revert Virtual Machine** dialog box, click **Revert**.
4. Repeat steps 2 and 3 for 10961B-LON-CL1.

**Results:** After completing this exercise, you will have created and run a scheduled job, and retrieved the results from the job.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 12: Using Profiles and Advanced Windows PowerShell Techniques

# Lab: Practicing Advanced Techniques

### Exercise 1: Using Advanced Techniques

► Task 1: Use Advanced Operators

1. Ensure you are signed into 10961B-LON-CL1 as ADATUM\Administrator with password Pa\$\$w0rd.
2. Open the Administrator Windows PowerShell console.
3. Run the following command and answer Yes to the prompt by typing Y.

```
Set-ExecutionPolicy RemoteSigned
```

4. Run the file **E:\Mod12\Labfiles\Lab12.ps1** by typing the below and pressing Enter  

```
. E:\Mod12\Labfiles\Lab12.ps1
```

Notice that there are a period and a space before the **E:\**

5. Does **\$ServiceNames** include the value **WinRM**?

```
$ServiceNames -contains "WinRM"
```

6. Does **\$ServiceNames** include the value **Spooler**?

```
"Spooler" -in $ServiceNames
```

7. What is the second value in the array **\$ComputerNames**?

```
$ComputerNames[1]
```

8. Produce a comma-separated string that contains the values in the array **\$ComputerNames**.

```
$ComputerNames -Join ","
```

9. Produce a tab-delimited string that contains the values in **\$ComputerNames**.

```
$ComputerNames -Join "`t"
```

10. In the variable **\$phrase**, replace the word **dog** with the word **gelding**.

```
$phrase -replace 'dog','gelding'
```

11. **\$List** contains a comma-separated list. Display the second value in this list.

```
($List -Split ',')[1]
```

► Task 2: Manipulate string values

1. If you have not already done so, in Task 1, run . **E:\Mod12\Labfiles\Lab12.ps1**
2. Display an all-uppercase version of the contents of **\$phrase**.

MCT USE ONLY. STUDENT USE PROHIBITED

```
$phrase.ToUpper()
```

3. Display an all-lowercase version of the contents of **\$phrase**.

```
$phrase.ToLower()
```

4. In **\$phrase**, replace the value **over** with the value **around**, and display an all-uppercase version of the result.

```
$phrase.Replace('over', 'around').ToUpper()
```

5. Display the sixth through the eighth characters in **\$list**.

```
$list.Substring(5, 3)
```

6. Display the contents of **\$padded** so that there are no additional spaces before or after **value**.

```
$padded.Trim()
```

7. Display the number of characters in **\$phrase**.

```
$phrase.Length
```

8. Display **True** or **False**, depending on whether the contents of **\$phrase** starts with the value **The**.

```
$phrase.StartsWith('The')
```

9. Display the **contents** of **\$unpadded** so that 10 additional spaces are added before **value**.

```
$unpadded.PadLeft(10)
```

### ► Task 3: Manipulate date values

1. Store the current date in the variable **\$today**.

```
$today = Get-Date
```

2. Display the full date of 10 days ago.

```
$today.AddDays(-10)
```

3. Display the full date of 30 days from today.

```
$today.AddDays(30)
```

4. Display the current hour.

```
$today.Hour
```

5. Display the current month.

```
$today.Month
```

6. Display today's date as a short date string.

```
$today.ToShortDateString()
```

7. Display today's date in Universal time.

```
$today.ToUniversalTime()
```

8. Display the current time as a long time string.

```
$today.ToString("yyyy-MM-dd HH:mm:ss")
```

**Results:** After completing this exercise, you will have practice with using several Windows PowerShell techniques.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 2: Using Alternative Credentials

### ► Task 1: Create a credential object

1. To create the credential object, run:

```
$admin = Get-Credential -Credential ADATUM\Administrator
```

2. In the dialog box, enter **Pa\$\$w0rd** and press Enter.

### ► Task 2: Run a Command By Using a Credential

1. To use the credential object to invoke the command, run:

```
Invoke-Command -ComputerName LON-DC1 -Credential $admin -ScriptBlock { Get-ADUser -Filter * | Export-Csv C:\Users.csv }
```

2. Switch to the LON-DC1 virtual machine and type **cd\** at the command prompt and then press Enter.
3. Then type **dir** and press Enter.
4. In the list if files returned verify there is a file present called **C:\Users.csv**
5. Open the file by typing **Notepad** and pressing Enter
6. Then in Notepad go to **File > Open**, change the **Files of types** to **All Files**, and select **C:\Users.csv** to display its contents
7. Verify the file contains a list of user account details

**Results:** After completing this exercise, you will have practiced how to use alternative credentials.

## Exercise 3: Create a Profile Script

### ► Task 1: Create a Profile Script

1. On the LON-CL1 virtual machine, in a console window, run **ISE**
2. In the ISE window, press **Ctrl+N**.
3. In the Script Pane, enter the following command:

```
Cd c:\  
$cred = Get-Credential -Credential ADATUM\Administrator
```

4. Press **Ctrl+S**.
5. In the tree view on the left, expand **This PC** and then expand and select **Documents**.
6. If the **WindowsPowerShell** folder does not exist within the **Documents** folder, click **New folder**. Type **WindowsPowerShell** for the folder name, and press Enter.
7. Double-click the **WindowsPowerShell** folder.
8. In the **File name** text box, type **Profile.ps1** and press Enter.

### ► Task 2: Define default values for a command

1. In the Integrated Scripting Environment (ISE) window, click the tab for your **Profile.ps1** script.
2. At the end of the script, add the following:

```
$PSDefaultParameterValues=@{"Get-EventLog:LogName"="Security";"Get-  
EventLog>Newest"=10}
```

3. Press **Ctrl+S**.

### ► Task 3: Test the profile script

1. Open the Windows PowerShell console If it is not already open. i.e. in the taskbar, right-click the Windows PowerShell icon and select **Run as Administrator**.

(If the Windows PowerShell icon doesn't exist in the taskbar you can open the **Start Screen** and type Power, then right clicking the resultant **Windows PowerShell** icon and selecting the **Pin to taskbar** option.)

2. In the credential dialog box, enter **Pa\$\$w0rd** and press Enter.
3. Verify that the prompt is **C:\**.
4. Run **\$cred** and verify that a credential is listed.
5. Run **Get-EventLog**. Verify that 10 entries from the Security log are shown.

### ► Task 4: To prepare for another module

If you are intending continuing to another module to perform labs, once you have finished this lab, revert the virtual machines to their initial state. To do this, perform the following steps:

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

**Results:** After completing this exercise, you will have created a profile script.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes

MCT USE ONLY. STUDENT USE PROHIBITED

---

## Notes