

**Microsoft®**

MCT USE ONLY. STUDENT USE PROHIBITED

OFFICIAL MICROSOFT LEARNING PRODUCT

10325A

Automating Administration with Windows  
PowerShell® 2.0

MCT USE ONLY. STUDENT USE PROHIBITED

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.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Product Number: 10325A

Part Number: X17-46558

Released: 09/2010

# **MICROSOFT LICENSE TERMS**

## **OFFICIAL MICROSOFT LEARNING PRODUCTS - TRAINER EDITION –**

### **Pre-Release and Final Release Versions**

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the Licensed Content named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this Licensed Content, unless other terms accompany those items. If so, those terms apply.

**By using the Licensed Content, you accept these terms. If you do not accept them, do not use the Licensed Content.**

**If you comply with these license terms, you have the rights below.**

#### **1. DEFINITIONS.**

- a. **"Academic Materials"** means the printed or electronic documentation such as manuals, workbooks, white papers, press releases, datasheets, and FAQs which may be included in the Licensed Content.
- b. **"Authorized Learning Center(s)"** means a Microsoft Certified Partner for Learning Solutions location, an IT Academy location, or such other entity as Microsoft may designate from time to time.
- c. **"Authorized Training Session(s)"** means those training sessions authorized by Microsoft and conducted at or through Authorized Learning Centers by a Trainer providing training to Students solely on Official Microsoft Learning Products (formerly known as Microsoft Official Curriculum or "MOC") and Microsoft Dynamics Learning Products (formerly known as Microsoft Business Solutions Courseware). Each Authorized Training Session will provide training on the subject matter of one (1) Course.
- d. **"Course"** means one of the courses using Licensed Content offered by an Authorized Learning Center during an Authorized Training Session, each of which provides training on a particular Microsoft technology subject matter.
- e. **"Device(s)"** means a single computer, device, workstation, terminal, or other digital electronic or analog device.
- f. **"Licensed Content"** means the materials accompanying these license terms. The Licensed Content may include, but is not limited to, the following elements: (i) Trainer Content, (ii) Student Content, (iii) classroom setup guide, and (iv) Software. There are different and separate components of the Licensed Content for each Course.
- g. **"Software"** means the Virtual Machines and Virtual Hard Disks, or other software applications that may be included with the Licensed Content.
- h. **"Student(s)"** means a student duly enrolled for an Authorized Training Session at your location.
- i. **"Student Content"** means the learning materials accompanying these license terms that are for use by Students and Trainers during an Authorized Training Session. Student Content may include labs, simulations, and courseware files for a Course.
- j. **"Trainer(s)"** means a) a person who is duly certified by Microsoft as a Microsoft Certified Trainer and b) such other individual as authorized in writing by Microsoft and has been engaged by an Authorized Learning Center to teach or instruct an Authorized Training Session to Students on its behalf.
- k. **"Trainer Content"** means the materials accompanying these license terms that are for use by Trainers and Students, as applicable, solely during an Authorized Training Session. Trainer Content may include Virtual Machines, Virtual Hard Disks, Microsoft PowerPoint files, instructor notes, and demonstration guides and script files for a Course.
- l. **"Virtual Hard Disks"** means Microsoft Software that is comprised of virtualized hard disks (such as a base virtual hard disk or differencing disks) for a Virtual Machine that can be loaded onto a single computer or other device in order to allow end-users to run multiple operating systems concurrently. For the purposes of these license terms, Virtual Hard Disks will be considered "Trainer Content".
- m. **"Virtual Machine"** means a virtualized computing experience, created and accessed using Microsoft® Virtual PC or Microsoft® Virtual Server software that consists of a virtualized hardware environment, one or more Virtual Hard Disks,

MCT USE ONLY. STUDENT USE PROHIBITED

and a configuration file setting the parameters of the virtualized hardware environment (e.g., RAM). For the purposes of these license terms, Virtual Hard Disks will be considered "Trainer Content".

- n. "you" means the Authorized Learning Center or Trainer, as applicable, that has agreed to these license terms.

## 2. OVERVIEW.

**Licensed Content.** The Licensed Content includes Software, Academic Materials (online and electronic), Trainer Content, Student Content, classroom setup guide, and associated media.

**License Model.** The Licensed Content is licensed on a per copy per Authorized Learning Center location or per Trainer basis.

## 3. INSTALLATION AND USE RIGHTS.

### a. Authorized Learning Centers and Trainers: For each Authorized Training Session, you may:

- i. either install individual copies of the relevant Licensed Content on classroom Devices only for use by Students enrolled in and the Trainer delivering the Authorized Training Session, provided that the number of copies in use does not exceed the number of Students enrolled in and the Trainer delivering the Authorized Training Session, OR
  - ii. install one copy of the relevant Licensed Content on a network server only for access by classroom Devices and only for use by Students enrolled in and the Trainer delivering the Authorized Training Session, provided that the number of Devices accessing the Licensed Content on such server does not exceed the number of Students enrolled in and the Trainer delivering the Authorized Training Session.
  - iii. and allow the Students enrolled in and the Trainer delivering the Authorized Training Session to use the Licensed Content that you install in accordance with (ii) or (ii) above during such Authorized Training Session in accordance with these license terms.
- i. Separation of Components. The components of the Licensed Content are licensed as a single unit. You may not separate the components and install them on different Devices.
  - ii. Third Party Programs. The Licensed Content may contain third party programs. These license terms will apply to the use of those third party programs, unless other terms accompany those programs.

### b. Trainers:

- i. Trainers may Use the Licensed Content that you install or that is installed by an Authorized Learning Center on a classroom Device to deliver an Authorized Training Session.
- ii. Trainers may also Use a copy of the Licensed Content as follows:
  - A. Licensed Device. The licensed Device is the Device on which you Use the Licensed Content. You may install and Use one copy of the Licensed Content on the licensed Device solely for your own personal training Use and for preparation of an Authorized Training Session.
  - B. Portable Device. You may install another copy on a portable device solely for your own personal training Use and for preparation of an Authorized Training Session.

## 4. PRE-RELEASE VERSIONS.

If this is a pre-release ("beta") version, in addition to the other provisions in this agreement, these terms also apply:

- a. **Pre-Release Licensed Content.** This Licensed Content is a pre-release version. It may not contain the same information and/or work the way a final version of the Licensed Content will. We may change it for the final, commercial version. We also may not release a commercial version. You will clearly and conspicuously inform any Students who participate in each Authorized Training Session of the foregoing; and, that you or Microsoft are under no obligation to provide them with any further content, including but not limited to the final released version of the Licensed Content for the Course.
- b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, 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, Licensed Content, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.
- c. **Confidential Information.** The Licensed Content, including any viewer, user interface, features and documentation that may be included with the Licensed Content, is confidential and proprietary to Microsoft and its suppliers.

- i. **Use.** For five years after installation of the Licensed Content or its commercial release, whichever is first, you may not disclose confidential information to third parties. You may disclose confidential information only to your employees and consultants who need to know the information. You must have written agreements with them that protect the confidential information at least as much as this agreement.
- ii. **Survival.** Your duty to protect confidential information survives this agreement.
- iii. **Exclusions.** You may disclose confidential information in response to a judicial or governmental order. You must first give written notice to Microsoft to allow it to seek a protective order or otherwise protect the information. Confidential information does not include information that
  - becomes publicly known through no wrongful act;
  - you received from a third party who did not breach confidentiality obligations to Microsoft or its suppliers; or
  - you developed independently.
- d. **Term.** The term of this agreement for pre-release versions is (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) the commercial release of the final release version of the Licensed Content, whichever is first ("beta term").
- e. **Use.** You will cease using all copies of the beta version upon expiration or termination of the beta term, and will destroy all copies of same in the possession or under your control and/or in the possession or under the control of any Trainers who have received copies of the pre-released version.
- f. **Copies.** Microsoft will inform Authorized Learning Centers if they may make copies of the beta version (in either print and/or CD version) and distribute such copies to Students and/or Trainers. If Microsoft allows such distribution, you will follow any additional terms that Microsoft provides to you for such copies and distribution.

## 5. ADDITIONAL LICENSING REQUIREMENTS AND/OR USE RIGHTS.

### a. Authorized Learning Centers and Trainers:

- i. Software.
- ii. **Virtual Hard Disks.** The Licensed Content may contain versions of Microsoft XP, Microsoft Windows Vista, Windows Server 2003, Windows Server 2008, and Windows 2000 Advanced Server and/or other Microsoft products which are provided in Virtual Hard Disks.

#### A. If the Virtual Hard Disks and the labs are launched through the Microsoft Learning Lab Launcher, then these terms apply:

**Time-Sensitive Software.** If the Software is not reset, it will stop running based upon the time indicated on the install of the Virtual Machines (between 30 and 500 days after you install it). You will not receive notice before it stops running. You may not be able to access data used or information saved with the Virtual Machines when it stops running and may be forced to reset these Virtual Machines to their original state. You must remove the Software from the Devices at the end of each Authorized Training Session and reinstall and launch it prior to the beginning of the next Authorized Training Session.

#### B. If the Virtual Hard Disks require a product key to launch, then these terms apply:

Microsoft will deactivate the operating system associated with each Virtual Hard Disk. Before installing any Virtual Hard Disks on classroom Devices for use during an Authorized Training Session, you will obtain from Microsoft a product key for the operating system software for the Virtual Hard Disks and will activate such Software with Microsoft using such product key.

#### C. These terms apply to all Virtual Machines and Virtual Hard Disks:

**You may only use the Virtual Machines and Virtual Hard Disks if you comply with the terms and conditions of this agreement and the following security requirements:**

- You may not install Virtual Machines and Virtual Hard Disks on portable Devices or Devices that are accessible to other networks.
- You must remove Virtual Machines and Virtual Hard Disks from all classroom Devices at the end of each Authorized Training Session, except those held at Microsoft Certified Partners for Learning Solutions locations.

- You must remove the differencing drive portions of the Virtual Hard Disks from all classroom Devices at the end of each Authorized Training Session at Microsoft Certified Partners for Learning Solutions locations.
- You will ensure that the Virtual Machines and Virtual Hard Disks are not copied or downloaded from Devices on which you installed them.
- You will strictly comply with all Microsoft instructions relating to installation, use, activation and deactivation, and security of Virtual Machines and Virtual Hard Disks.
- You may not modify the Virtual Machines and Virtual Hard Disks or any contents thereof.
- You may not reproduce or redistribute the Virtual Machines or Virtual Hard Disks.

**ii. Classroom Setup Guide.** You will assure any Licensed Content installed for use during an Authorized Training Session will be done in accordance with the classroom set-up guide for the Course.

**iii. Media Elements and Templates.** You may allow Trainers and Students to use images, clip art, animations, sounds, music, shapes, video clips and templates provided with the Licensed Content solely in an Authorized Training Session. If Trainers have their own copy of the Licensed Content, they may use Media Elements for their personal training use.

**iv. iv Evaluation Software.** Any Software that is included in the Student Content designated as "Evaluation Software" may be used by Students solely for their personal training outside of the Authorized Training Session.

**b. Trainers Only:**

**i. Use of PowerPoint Slide Deck Templates.** The Trainer Content may include Microsoft PowerPoint slide decks. Trainers may use, copy and modify the PowerPoint slide decks only for providing an Authorized Training Session. If you elect to exercise the foregoing, you will agree or ensure Trainer agrees: (a) that modification of the slide decks will not constitute creation of obscene or scandalous works, as defined by federal law at the time the work is created; and (b) to comply with all other terms and conditions of this agreement.

**ii. Use of Instructional Components in Trainer Content.** For each Authorized Training Session, Trainers may customize and reproduce, in accordance with the MCT Agreement, those portions of the Licensed Content that are logically associated with instruction of the Authorized Training Session. If you elect to exercise the foregoing rights, you agree or ensure the Trainer agrees: (a) that any of these customizations or reproductions will only be used for providing an Authorized Training Session and (b) to comply with all other terms and conditions of this agreement.

**iii. Academic Materials.** If the Licensed Content contains Academic Materials, you may copy and use the Academic Materials. You may not make any modifications to the Academic Materials and you may not print any book (either electronic or print version) in its entirety. If you reproduce any Academic Materials, you agree that:

- The use of the Academic Materials will be only for your personal reference or training use
- You will not republish or post the Academic Materials on any network computer or broadcast in any media;
- You will include the Academic Material's original copyright notice, or a copyright notice to Microsoft's benefit in the format provided below:

**Form of Notice:**

© 2010 Reprinted for personal reference use only with permission by Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the US and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

**6. INTERNET-BASED SERVICES.** Microsoft may provide Internet-based services with the Licensed Content. It may change or cancel them at any time. You may not use these services in any way that could harm them or impair anyone else's use of them. You may not use the services to try to gain unauthorized access to any service, data, account or network by any means.

**7. 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 allow you to use it in certain ways. You may not

- install more copies of the Licensed Content on classroom Devices than the number of Students and the Trainer in the Authorized Training Session;
- allow more classroom Devices to access the server than the number of Students enrolled in and the Trainer delivering the Authorized Training Session if the Licensed Content is installed on a network server;
- copy or reproduce the Licensed Content to any server or location for further reproduction or distribution;
- disclose the results of any benchmark tests of the Licensed Content **to any third party without Microsoft's prior written approval**;
- work around any technical limitations in the Licensed Content;
- reverse engineer, decompile or disassemble the Licensed Content, except and only to the extent that applicable law expressly permits, despite this limitation;
- make more copies of the Licensed Content than specified in this agreement or allowed by applicable law, despite this limitation;
- publish the Licensed Content for others to copy;
- transfer the Licensed Content, in whole or in part, to a third party;
- access or use any Licensed Content for which you (i) are not providing a Course and/or (ii) have not been authorized by Microsoft to access and use;
- rent, lease or lend the Licensed Content; or
- use the Licensed Content for commercial hosting services or general business purposes.
- Rights to access the server software that may be included with the Licensed Content, including the Virtual Hard Disks does not give you any right to implement Microsoft patents or other Microsoft intellectual property in software or devices that may access the server.

**8. 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).

**9. NOT FOR RESALE SOFTWARE/LICENSED CONTENT.** You may not sell software or Licensed Content **marked as "NFR"** or **"Not for Resale."**

**10. ACADEMIC EDITION.** You must be a **"Qualified Educational User"** to use Licensed Content **marked as "Academic Edition"** or **"AE."** If you do not know whether you are a Qualified Educational User, visit [www.microsoft.com/education](http://www.microsoft.com/education) or contact the Microsoft affiliate serving your country.

**11. TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of these license terms. In the event your status as an Authorized Learning Center or Trainer a) expires, b) is voluntarily terminated by you, and/or c) is terminated by Microsoft, this agreement shall automatically terminate. Upon any termination of this agreement, you must destroy all copies of the Licensed Content and all of its component parts.

**12. ENTIRE AGREEMENT. This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the Licensed Content and support services.**

**13. 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.

**14. 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.

**15. DISCLAIMER OF WARRANTY.** The Licensed Content is licensed "as-is." You bear the risk of using it.

Microsoft 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 excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

**16. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO U.S. \$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, software, 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 connaît 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.

# 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

## Acknowledgements

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.

This course was designed and written by **Don Jones** and **Greg Shields** of Concentrated Technology, LLC. Don and Greg have more than thirty years of combined experience in administering Windows-based environments and have previously written and lectured on VBScript and other automation technologies. They maintain a firm focus on administration, rather than software development, helping administrators work more efficiently and effectively. They would like to thank Marco Shaw, Kirk Munro, Steven Murawski, and Jeffery Hicks for their work in writing the labs and lab answer keys for this course, and Christopher Gannon for his project management work. They are especially appreciative to the members of the Windows PowerShell product team for their early feedback on the course design and content, and thankful for the team at Microsoft Learning who helped make this course a reality.

Technical review was carried out by Thomas Lee. Thomas is an IT consultant and trainer living in the UK. A forty-year veteran of the IT industry, Thomas has been active in the Microsoft arena since Microsoft first released MS-DOS in 1981. He has worked for a number of large firms, including Accenture, Microsoft, and Global Knowledge. He has spoken at IT conferences around the globe, including U.S./UK and other TechEd conferences and for many years has written for a number of UK and U.S. magazines.

Thomas was one of the first to see the potential of PowerShell when Microsoft released a beta during 1993 and the first person in the world to blog about it! Thomas teaches PowerShell around the world and has worked with several firms to help them to adopt PowerShell and maximize its potential. He also maintains two blogs (*Under The Stairs* (<http://tfl09.blogspot.com>), largely focused on PowerShell, and *PowerShell Scripts Blog* (<http://shscripts.blogspot.com>), a blog that publishes PowerShell scripts). Thomas is highly active in the PowerShell community. He is a director of PowerShellCommunity.Org, is a moderator for The Scripting Guys Forum. He has been awarded Microsoft's MVP award fourteen times over the last fifteen years in recognition of his significant community efforts.

Thomas lives in the English countryside with his wife, Susan, and daughter, Rebecca.

# Contents

## Module 1: Fundamentals for Using Microsoft® PowerShell® v2

<b>Lesson 1:</b> Windows PowerShell Technology Background and Overview	1-3
<b>Lesson 2:</b> Windows PowerShell As an Interactive Command-Line Shell	1-14
<b>Lab A:</b> Using Windows PowerShell As an Interactive Command-Line Shell	1-42
<b>Lesson 3:</b> Using the Windows PowerShell Pipeline	1-50
<b>Lab B:</b> Using the Windows PowerShell Pipeline	1-70

## Module 2: Understanding and Using the Formatting System

<b>Lesson 1:</b> Understanding the Formatting System	2-3
<b>Lesson 2:</b> Using the Formatting System	2-17
<b>Lab:</b> Using the Formatting Subsystem	2-29

## Module 3: Core Windows PowerShell® Cmdlets

<b>Lesson 1:</b> Core Cmdlets for Everyday Use	3-3
<b>Lab A:</b> Using the Core Cmdlets	3-22
<b>Lesson 2:</b> Comparison Operators, Pipeline Filtering, and Object Enumeration	3-29
<b>Lab B:</b> Filtering and Enumerating Objects in the Pipeline	3-42
<b>Lesson 3:</b> Advanced Pipeline Techniques	3-49
<b>Lab C:</b> Using Pipeline Parameter Binding	3-63

## Module 4: Windows® Management Instrumentation

<b>Lesson 1:</b> Windows Management Instrumentation Overview	4-3
<b>Lesson 2:</b> Using Windows Management Instrumentation	4-22
<b>Lab A:</b> Using Windows Management Instrumentation in Windows PowerShell	4-35

## Module 5: Automating Active Directory® Administration

<b>Lesson 1:</b> Active Directory Automation Overview	5-3
<b>Lesson 2:</b> Managing Users and Groups	5-10
<b>Lab A:</b> Managing Users and Groups	5-14
<b>Lesson 3:</b> Managing Computers and Other Directory Objects	5-19
<b>Lab B:</b> Managing Computers and Other Directory Objects	5-22

## Module 6: Windows PowerShell® Scripts

<b>Lesson 1:</b> Script Security	6-3
<b>Lesson 2:</b> Basic Scripts	6-15
<b>Lesson 3:</b> Parameterized Scripts	6-22
<b>Lab:</b> Writing Windows PowerShell Scripts	6-30

**Module 7: Background Jobs and Remote Administration**

<b>Lesson 1:</b> Working with Background Jobs	7-3
<b>Lab A:</b> Working with Background Jobs	7-15
<b>Lesson 2:</b> Using Windows PowerShell Remoting	7-23
<b>Lab B:</b> Using Windows PowerShell Remoting	7-55

**Module 8: Advanced Windows PowerShell® Tips and Tricks**

<b>Lesson 1:</b> Using Profiles	8-3
<b>Lesson 2:</b> Reusing Scripts and Functions	8-11
<b>Lesson 3:</b> Writing Comment-Based Help	8-24
<b>Lab:</b> Advanced PowerShell Tips and Tricks	8-30

**Module 9: Automating Windows Server 2008 R2 Administration**

<b>Lesson 1:</b> Windows Server 2008 R2 Modules Overview	9-3
<b>Lesson 2:</b> Server Manager Cmdlets Overview	9-11
<b>Lab A:</b> Using the Server Manager Cmdlets	9-13
<b>Lesson 3:</b> Group Policy Cmdlets Overview	9-19
<b>Lab B:</b> Using the Group Policy Cmdlets	9-22
<b>Lesson 4:</b> Troubleshooting Pack Overview	9-27
<b>Lab C:</b> Using the Troubleshooting Pack Cmdlets	9-29
<b>Lesson 5:</b> Best Practices Analyzer Cmdlets Overview	9-33
<b>Lab D:</b> Using the Best Practices Analyzer Cmdlets	9-35
<b>Lesson 6:</b> IIS Cmdlets Overview	9-39
<b>Lab E:</b> Using the IIS Cmdlets	9-44

**Module 10: Reviewing and Reusing Windows PowerShell® Scripts**

<b>Lesson 1:</b> Example Script Overview	10-3
<b>Lesson 2:</b> Understanding Scripts	10-8

**Module 11: Writing Your Own Windows PowerShell® Scripts**

<b>Lesson 1:</b> Variables, Arrays, Escaping, and More Operators	11-3
<b>Lab A:</b> Using Variables and Arrays	11-30
<b>Lesson 2:</b> Scope	11-39
<b>Lesson 3:</b> Scripting Constructs	11-52
<b>Lab B:</b> Using Scripting Constructs	11-74
<b>Lesson 4:</b> Error Trapping and Handling	11-81
<b>Lab C:</b> Error Trapping and Handling	11-103
<b>Lesson 5:</b> Debugging Techniques	11-111
<b>Lab D:</b> Debugging a Script	11-125
<b>Lesson 6:</b> Modularization	11-130
<b>Lab E:</b> Modularization	11-154

**Lab Answer Keys**

# About This Course

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

## Course Description

This five-day, instructor-led course offers you the knowledge and helps you develop the skills you need to automate administrative tasks using Windows PowerShell® version 2. The course describes core features and capabilities of Windows PowerShell version 2, using Windows Server® 2008 R2 as the example software environment. However, this course is not intended to provide comprehensive coverage of either Windows PowerShell version 2 features or Windows Server 2008 R2 features. The most important goal of this course is to teach you to *learn how to use Windows PowerShell on your own*. With that goal in mind, this course will teach you how to use core Windows PowerShell functionality that you can use no matter what administrative tasks you attempt. And this course will teach you how to use Windows PowerShell discoverability features; these will help you teach yourself how to use whatever Windows PowerShell-integrated products you need to manage.

## Audience

The primary audience for this course is IT professionals who will use Windows PowerShell to automate administrative tasks in Windows Server 2008 R2 and in other Microsoft® products that support Windows PowerShell version 2.

The students for this course are typically responsible for managing their current servers. They should have a minimum of 1.5 years of experience working with Microsoft Windows Server 2008 as a server administrator. Prior experience with scripting is not expected. However, familiarity with command-line concepts and tools is highly recommended.

Students for this course may also be Windows Server Technical Specialists (administrators) and be responsible for hands-on deployment and day-to-day management of Windows-based servers and Windows-based client computers for enterprise organizations. These administrators manage file and print servers, network infrastructure servers, Web servers, and IT application servers, as well as support client computers. They use graphical administration tools as their primary interface but want to use Windows PowerShell cmdlets and write PowerShell scripts for routine tasks and bulk operations. They conduct most server management tasks remotely by using Remote Desktop, Server Manager, or other administration tools installed on their local workstations.

## Student Prerequisites

This course requires that you meet the following prerequisites:

- Be able to implement and maintain Windows® networking technologies.
- Be able to administer and troubleshoot Windows Server technologies.
- Be able to administer Active Directory® and Group Policy.
- Be able to administer Windows Server 2008 Web application server technologies.

## Course Objectives

After completing this course, students will be able to:

- Discover, locate, and learn how to use Windows PowerShell commands, including built-in commands as well as add-in commands.

- Use Windows PowerShell to retrieve, create, modify, and remove management information from Windows Server 2008 R2 and other Microsoft products that integrate with Windows PowerShell.
- Use Windows PowerShell to manipulate, filter, sort, group, and format management information.
- Use Windows PowerShell to write complex scripts and to debug and modularize those scripts.
- Use Windows PowerShell to complete key administrative tasks in Active Directory, remote client computer troubleshooting, and more.

## Course Outline

This section provides an outline of the course:

**Module 1,** "Fundamentals for Using Windows PowerShell v2," describes Windows PowerShell and introduces key skills, including the use of command-line commands, navigating the file system, and accessing built-in help for a command. This module also explains and demonstrates the Windows PowerShell pipeline.

**Module 2,** "Understanding and Using the Formatting System," describes how to manipulate the final output of a series of commands to create formatted files, screen displays, and other output.

**Module 3,** "Core Windows PowerShell Cmdlets," describes some of the core commands that Windows PowerShell provides for manipulating information in the pipeline, and it covers pipeline parameter binding. This module also covers comparison operators and pipeline filtering, and it introduces objects.

**Module 4,** "Windows Management Instrumentation," describes the structure and use of Windows Management Instrumentation (WMI). This module also describes advanced WMI techniques, including the use of WMI methods and events.

**Module 5,** "Automating Active Directory Administration," describes how to use add-in commands to manage and manipulate Active Directory objects. This module builds on skills learned in previous modules to enable students to discover how to complete specified Active Directory tasks on their own, without complete directions from the instructor.

**Module 6,** "Windows PowerShell Scripts," describes how to turn commands into scripts that enable easier repeated command execution. This module also addresses security concerns around script files and script execution.

**Module 7,** "Background Jobs and Remote Administration," describes how to enable and use Windows PowerShell for remote computer administration and how to create jobs that complete administrative tasks in the background.

**Module 8,** "Advanced Windows PowerShell Tips and Tricks," describes how to create profile scripts and script modules and how to write comment-based help for scripts and functions.

**Module 9,** "Automating Windows Server 2008 R2 Administration," directs students to complete numerous Windows Server 2008 R2 administrative tasks on their own, with minimal instructor direction or guidance. Tasks include automating Server Manager, Group Policy, IIS, troubleshooting, and Best Practices Analyzer.

**Module 10,** "Reviewing and Reusing Windows PowerShell Scripts," describes how to take a script that someone else has written, review that script to understand what it does, and identify areas of that script that may need to be modified to run in their environment.

**Module 11,** "Writing Your Own Windows PowerShell Scripts," describes how to use the Windows PowerShell scripting language to automate complex multistep administrative tasks and processes. This module also describes variables and arrays within the shell.

**Appendix,** "Additional Windows PowerShell Features," describes additional Windows PowerShell version 2 features and capabilities and is intended to give students a starting point for independent exploration and learning. This appendix includes examples of how to use selected technologies and provides students with suggestions for further exploration and experimentation.

## 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.
- **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 finger tips 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®, 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@microsoftcourseware.com](mailto:support@microsoftcourseware.com). To inquire about the Microsoft Certification Program, send e-mail to [mchelp@microsoft.com](mailto:mchelp@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 Windows PowerShell deployed on Windows Server 2008 R2 and Windows 7 to perform the labs. The virtual machines used in the labs and demos will run in Hyper-V and each student will require one host computer to perform the labs.

**Important:** The labs in this course are not modular—that is, there are no dependencies among the labs in each module. At the end of each lab, you may shut down and restart the virtual machines if you wish, but this is not required. Do not create snapshots of the virtual machines in the labs unless instructed to do so.

The following table shows the role of each virtual machine used in this course:

Virtual machine	Role
10325A-LON-DC1	Domain controller in the Contoso.com domain. This virtual machine is required for all labs, and this is the virtual machine that you will usually use to complete the labs and to run any demonstration scripts or commands.
10352A-LON-SVR1	Member server in the Contoso.com domain.
10325A-LON-SVR2	Member server in the Contoso.com domain.
10325A-LON-CLIENT	Windows 7 client computer in the Contoso.com domain.

## Software Configuration

The following software is installed in host and virtual machines:

- Windows Server 2008
- Windows Server 2008 R2
- Windows 7

## Classroom Setup

Each student in this course will use one host computer. Each classroom computer will have the same virtual machines configured in the same way.

## 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. Each student will use one host computer with the following hardware configuration.

- 64 bit Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120 GB hard disks 7200 RM SATA or better
- 4 GB RAM expandable to 8 GB
- DVD drive
- Network adapter
- Super VGA (SVGA) 17-inch monitor
- Microsoft mouse or compatible pointing device

In addition, the instructor computer must be connected to a projection display device that supports SVGA 1024 x 768 pixels, 16-bit colors.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 1

## Fundamentals for Using Microsoft® Windows PowerShell® v2

### Contents:

<b>Lesson 1:</b> Windows PowerShell Technology Background and Overview	1-3
<b>Lesson 2:</b> Windows PowerShell As an Interactive Command-Line Shell	1-14
<b>Lab A:</b> Using Windows PowerShell As an Interactive Command-Line Shell	1-42
<b>Lesson 3:</b> Using the Windows PowerShell Pipeline	1-50
<b>Lab B:</b> Using the Windows PowerShell Pipeline	1-72

## Module Overview

- Install and customize Windows PowerShell v2
- Explain where Windows PowerShell v2 fits into the Microsoft Windows Management Framework
- Create and use PSDrives to manipulate files, folders, and other storage systems
- Locate, display, and interpret the built-in help for a cmdlet
- Extend the shell by using PSSnapins and modules
- Locate the cmdlet or cmdlets that can perform a specific task
- Use cmdlets within the shell pipeline
- Display the properties, methods, and other members of an element



Microsoft® Windows PowerShell® is a modern, standardized command-line shell that offers administrators more flexibility and choice in how they manage their Windows-based computers and products. Windows PowerShell is not a direct replacement for older technologies like Microsoft Visual Basic® Script Edition (VBScript), although the shell does have scripting capabilities, and does provide the same functionality as VBScript—and then some. Windows PowerShell is inspired by the best administrative tools and processes from Windows and non-Windows operating systems, and offers a uniquely Windows-centric administration model.

For perhaps the first time, Windows PowerShell offers a truly administrator-focused means of automating repetitive or time-consuming administrative tasks, without the need for the same complex programming or system scripting that was required by older technologies.

### Module Objectives

After completing this module, you will be able to:

- Install and customize Windows PowerShell v2.
- Explain where Windows PowerShell v2 fits into the Microsoft Windows® Management Framework.
- Create and use PSDrives to manipulate files, folders, and other storage systems.
- Locate, display, and interpret the built-in help for a cmdlet.
- Extend the shell by using PSSnapins and modules.
- Locate the cmdlet or cmdlets that can perform a specific task.
- Use cmdlets within the shell pipeline.
- Display the properties, methods, and other members of an object.

## Lesson 1

# Windows PowerShell Technology Background and Overview

- Describe how Windows PowerShell can act as both a command-line shell and a back-end engine for running administrative commands
- List the system requirements for Windows PowerShell v2
- Install and customize Windows PowerShell v2



Windows PowerShell isn't just a new scripting language. It's actually a brand new way to manage Windows and Windows-based products, and a new way for Microsoft to think about administrators and how they work. Many of the things you can do in Windows PowerShell will seem immediately familiar, such as listing the contents of a directory on disk. However, Windows PowerShell is usually doing something very different under the hood. Being successful with Windows PowerShell requires you to have a basic understanding of the shell's operational strategy, its technology background, and its design goals.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe how Windows PowerShell can act as both a command-line shell and a back-end engine for running administrative commands.
- List the system requirements for Windows PowerShell v2.
- Install and customize Windows PowerShell v2.

MCT USE ONLY. STUDENT USE PROHIBITED

## The Purpose of Windows PowerShell

- Windows PowerShell is not a scripting language
  - At least, it is not *only* a scripting language
- PowerShell is an engine designed to execute commands that perform administrative tasks, for example
  - Creating user accounts
  - Configuring services
  - Deleting mailboxes
- PowerShell provides a foundation upon which Microsoft's GUI-based administrative tools can rest
  - Actions can be accomplished in its command-line console
  - Actions can also be invoked within GUIs by executing PowerShell commands in the background

### Key Points

Windows PowerShell is not a scripting language, or at least it is not *just* a scripting language. Windows PowerShell is an engine designed to execute commands that perform administrative tasks, such as creating new user accounts, configuring services, deleting mailboxes, and so forth.

Windows PowerShell actually provides many ways in which you can tell it what commands to execute. You can, for example, manually type command names in a command-line console window. You can also type commands in an integrated scripting environment, (ISE) that offers a more graphically-rich command-line environment. If you are a software developer, you can embed Windows PowerShell within your own application, telling it to execute its commands in response to user actions such as clicking buttons or icons. You could also type a series of commands into a text file, and instruct the shell to execute the commands in that file.

In an ideal world, Windows PowerShell is a single, central source for administrative functionality. Ideally, you could use a graphical user interface (GUI) with buttons, icons, dialog boxes, and other elements that execute Windows PowerShell commands in the background. If the GUI didn't allow you to accomplish a task in exactly the way you wanted, you could choose to execute those same commands—in the order and way you prefer—directly in the command-line console, bypassing the GUI. Many Microsoft products are built in that exact way, including Microsoft Exchange Server 2007 and Microsoft Exchange Server 2010. Some products, like the Windows Server operating system, are not built in that way yet, although some specific components are. For example, the Active Directory Administrative Center in Windows Server 2008 R2 is built in this ideal way, meaning you can choose to use a GUI that runs Windows PowerShell commands in the background, or you can choose to run the commands yourself directly in the Windows PowerShell console or ISE.

This choice, to use commands directly, or to have commands run for you as part of a GUI, is part of what makes Windows PowerShell so compelling. With this shell, Microsoft recognizes and acknowledges that some tasks are easier to do in a GUI, especially tasks that you don't perform very often. A GUI can guide you through complex operations, and can help you understand your choices and options more easily.

However, Microsoft also recognizes that a GUI can be inefficient for tasks that you need to perform repeatedly, such as creating new user accounts. By building as much administrative functionality as possible in the form of Windows PowerShell commands, Microsoft lets *you* choose what's right for any given task: The ease-of-use of a GUI, or the power and customization of a command-line shell.

Over time, Windows PowerShell may replace other low-level administrative tools that you may have used. For example, Windows PowerShell can already supplant Visual Basic Script Edition (VBScript), because the shell has access to the same features that VBScript does, although in many cases the shell provides easier ways to accomplish the same tasks. Windows PowerShell may also replace your use of Windows Management Instrumentation (WMI). While WMI remains very useful, it can also be complex to use. Windows PowerShell can wrap task-specific commands around underlying WMI functionality. You're technically still using WMI, but doing so becomes easier because you can run an easier-to-use, task-based command. All of these things are happening now, and will continue to improve in the future as Windows PowerShell itself evolves.

MCT USE ONLY. STUDENT USE PROHIBITED

## Installing Windows PowerShell v2

- Windows PowerShell is pre-installed by default in Windows Server 2008 R2 and Windows 7
- Windows PowerShell is a Web download for Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008
- Windows PowerShell requires the Microsoft .NET Framework 2.0 with Service Pack 1
- The Windows PowerShell ISE requires the Microsoft .NET Framework 3.5 with Service Pack 1
- Windows PowerShell exposes newer functionality in newer versions of Windows
  - Older versions of Windows will have a slightly reduced feature set, because those versions do not contain the underlying functionality that a feature requires

### Key Points

Windows PowerShell v2 is pre-installed by default in Windows Server 2008 R2 and Windows 7. In Windows Server 2008 R2, you can optionally install the Windows PowerShell ISE, a graphically-oriented shell environment.

Windows PowerShell v2 is also available as a Web download for Windows XP, Windows Server® 2003, Windows Vista®, and Windows Server 2008. Windows PowerShell v2 is included in the Windows Management Framework Core, which also includes other related management technologies. The download can be found at <http://go.microsoft.com/fwlink/?LinkId=193574>; separate versions are available for different operating systems and architectures (32-bit and 64-bit). The download includes the Windows PowerShell ISE as well as the more traditional command-line console.

Windows PowerShell v2 can be installed on the following operating systems:

- Windows Server 2008 with Service Pack 1
- Windows Server 2008 with Service Pack 2
- Windows Server 2003 with Service Pack 2
- Windows Vista with Service Pack 2
- Windows Vista with Service Pack 1
- Windows XP with Service Pack 3
- Windows Embedded POSReady 2009
- Windows Embedded for Point of Service 1.1

Windows PowerShell v2 requires Microsoft .NET Framework 2.0 with Service Pack 1; the Windows PowerShell ISE requires Microsoft .NET Framework 3.5 with Service Pack 1. Note that Windows PowerShell can expose additional functionality and features from newer versions of Microsoft .NET Framework. Ideally, you should install the highest available version of the Framework to gain the most functionality.

Windows PowerShell also exposes new functionality in newer versions of Windows. That means you may have a slightly reduced feature set on older versions of Windows, simply because those older versions do not contain the underlying functionality that a feature requires.

**Note:** The information in this course applies to Windows PowerShell v2 running on all supported versions of Windows, unless specifically stated otherwise.

MCT USE ONLY. STUDENT USE PROHIBITED

## What Else Will You Need?

- PowerShell is designed to be extended through snap-ins and modules
  - Snap-ins and modules can be provided by Microsoft, or other third-party organizations
  - You can write your own snap-ins and modules
  - Snap-ins and modules tend to relate to particular products, roles, features, or technologies
  - Snap-ins and modules may have system requirements beyond PowerShell's own requirements
  - Snap-ins and modules will tend to have their own documentation and help
- Think about modules and snap-ins as similar to the MMC's snap-ins

### Key Points

Windows PowerShell is a core management framework, not unlike the Microsoft Management Console (MMC). You are probably aware that the MMC, by itself, is not useful for very many tasks. In order to make the MMC useful, you have to add one or more snap-ins. Those snap-ins provide product-specific and technology-specific functionality, such as enabling you to administer Active Directory or Exchange Server.

Windows PowerShell is similar. While it does include a number of useful capabilities, it is designed to be extended through snap-ins and modules. Windows PowerShell itself does not provide these modules or snap-ins; they are provided along with whatever product, role, feature, or technology that they relate to. In other words, if you want to use Windows PowerShell to manage Active Directory Domain Services, you need a module for that, and that module installs along with the Active Directory® Domain Services role. Some modules may also be installed along with the Windows Remote Server Administration Toolkit (RSAT), making it possible to obtain server-related modules on a client operating system like Windows 7.

Some modules may be available as standalone downloads. For example, the CodePlex Web site ([www.codeplex.com](http://www.codeplex.com)) hosts a number of third-party, open-source projects related to Windows PowerShell, many of which are packaged as snap-ins or modules. One of these is the popular Windows PowerShell Community Extensions, available at [www.codeplex.com/powershellcx](http://www.codeplex.com/powershellcx).

Note that some modules or snap-ins may have system requirements beyond Windows PowerShell's own requirements. Modules may require a specific version of Windows, or a specific version of Microsoft .NET Framework. The documentation for a given module or snap-in should indicate any system requirements above and beyond the shell's base requirements.

## Where Is Windows PowerShell Used?

- Windows PowerShell is used within a number of Microsoft and third-party products, including:
  - Microsoft Exchange Server 2007 and later
  - Microsoft System Center Data Protection Manager
  - Microsoft System Center Operations Manager
  - Microsoft System Center Virtual Machine Manager
  - Microsoft SQL Server 2008 and later

```
Get-Mailbox | Sort Size | Select -first 100 | Move-Mailbox Server2
```

### Key Points

Windows PowerShell is used within a number of Microsoft and third-party products. This list is constantly growing, and both Microsoft and third-party software developers often announce Windows PowerShell compatibility with new product versions.

Here is a partial list of current products that use Windows PowerShell. Note that this list does not include Windows Server 2008 R2 features that can be managed via Windows PowerShell; those features are discussed in Module 9 of this course.

- Microsoft Exchange Server 2007 and later
- Microsoft System Center Data Protection Manager
- Microsoft System Center Operations Manager
- Microsoft System Center Virtual Machine Manager
- Microsoft SQL Server 2008 and later

Windows PowerShell can make managing these products much easier, and can enable administrative tasks that might be difficult or impossible in the product's normal graphical user interface. For example, using Windows PowerShell commands with Exchange Server might allow you to get the largest mailboxes on a messaging server and move those to a new server. Such a command might look something like this:

```
Get-Mailbox | Sort Size | Select -first 100 | Move-Mailbox Server2
```

**Note:** This is a *pseudo-command*, meaning that this is not the exact syntax you would use to accomplish this task. This is intended as a simple illustration of what can be accomplished using Windows PowerShell. You will see numerous complete, real-life examples throughout this course.

This type of command is much more readable, and much easier to construct, than the complex scripts that were required with previous versions of Exchange Server prior to the introduction of Windows PowerShell.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Installing the ISE in Windows Server 2008 R2

- Learn how to install the Windows PowerShell ISE on Windows Server 2008 R2



### Key Points

In this demonstration, you will learn how to install the Windows PowerShell ISE on Windows Server 2008 R2.

### Demonstration steps:

1. Start and log on to virtual machine **LON-SVR2** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-SVR2** located in **E:\Mod01\Democode\Installing the ISE in Windows Server 2008 R2.txt**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Discussion: Benefits of Having the Shell Everywhere

- Will your organization consider a wide deployment of PowerShell?
- What questions or concerns might you have?



### Key Points

Windows PowerShell v2 is the first version of Windows PowerShell to be pre-installed, by default, in a Windows operating system, to be specific, Windows 7 and Windows Server 2008 R2. The goal behind this decision was to ensure that Windows PowerShell v2 is available on every computer in your environment.

Windows PowerShell v2 contains built-in *remoting* capabilities, allowing it to contact instances of Windows PowerShell on remote computers to perform management tasks. Some of the benefits of having Windows PowerShell on all of your client and server computers include:

- The ability to run interactive shell commands on remote computers.
- The ability to deploy scripts and commands to multiple remote computers in parallel.
- The ability to use Windows PowerShell-based logon scripts.

As a core part of the Windows operating system, updates and fixes to Windows PowerShell can be deployed in the same way as any operating system update or fix, including hot fixes and service packs, through systems like Windows Server Update Services (WSUS). This capability makes it easier to manage large Windows PowerShell deployments.

Also, many of the security configuration settings for Windows PowerShell's can be centrally configured in Group Policy objects, helping to streamline security and management of a wide Windows PowerShell deployment.

**Question:** Will your organization consider a wide deployment of Windows PowerShell v2? What questions or concerns might you have?

## Demonstration: Customizing the Shell

- Learn how to customize the appearance of the console window and the PowerShell ISE



### Key Points

In this demonstration, you will learn how to customize the appearance of the console window and the Windows PowerShell ISE.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Customizing the Shell.txt**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 2

# Windows PowerShell As an Interactive Command-Line Shell

- Explain the purpose and use of PSDrives
- Navigate the Windows file system and other PSDrives by using cmdlets
- Explain the relationship between cmdlets, aliases, and parameters
- Locate and interpret built-in help for a cmdlet, including detailed help and usage examples
- Extend the shell by using a PSSnapin or module
- Customize the output of a simple cmdlet
- Locate the correct cmdlet given a specific administrative task



While Windows PowerShell does include a simple and robust scripting language, one of the advantages of the shell is that you can be very effective *without* having to write complex scripts. Many powerful administrative tasks can be accomplished by simply using the shell to run commands interactively.

As an interactive shell, Windows PowerShell has many similarities to other command-line shells you may have used, including the Cmd.exe shell that has been included with Windows since Windows NT® 3.1. Many administrators with command-line experience are immediately comfortable using Windows PowerShell to perform familiar tasks, such as file and folder management, and those skills and techniques can quickly be expanded to include more complex and practical tasks.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and use of PSDrives.
- Navigate the Windows file system and other PSDrives by using cmdlets.
- Explain the relationship between cmdlets, aliases, and parameters.
- Locate and interpret built-in help for a cmdlet, including detailed help and usage examples.
- Extend the shell by using a PSSnapin or module.
- Customize the output of a simple cmdlet.
- Locate the correct cmdlet given a specific administrative task.

## Discussion: What Shell Commands Do You Know?

- What command would you use to perform each of the following tasks:
  - Change directories
  - List the files and subdirectories in a directory
  - Copy a file
  - Display the contents of a text file
  - Delete a file
  - Move a file
  - Rename a file
  - Create a new directory



### Key Points

Windows PowerShell was designed to be immediately usable by administrators who are already familiar with an older shell, including the Cmd.exe shell included with Windows, as well as common Unix shells. While some of the commands obviously work a little differently, you'll find many familiar commands available to you.

What command would you use to perform each of the following tasks:

- Change directories.
- List the files and subdirectories in a directory.
- Copy a file.
- Display the contents of a text file.
- Delete a file.
- Move a file.
- Rename a file.
- Create a new directory.

Windows PowerShell can also run most external commands that you may be familiar with, including:

- Ipconfig.exe
- Ping.exe
- Tracert.exe
- Nslookup.exe
- Pathping.exe
- Net.exe (for example, Net Use)

## Navigating the File System

- PowerShell recognizes many command names you already know
  - Most common file-and-folder management commands from both cmd.exe and common Unix shells are available
  - The parameters of the commands are often different
  - For example, the following command will generate an error:

```
dir /s
```

- External commands, such as ipconfig.exe, pathping.exe, and so on, continue to run exactly as they always have
  - The distinction is that external commands are external executables, whereas commands such as "dir" are internal (or intrinsic) commands

### Key Points

Windows PowerShell recognizes many of the command names that you are probably already familiar with, including Cd, Dir, Ls, Cat, Type, MkDir, RmDir, Rm, Del, Cp, Copy, Move, and so on. Most common file-and-folder management commands from both Cmd.exe (which uses the MS-DOS command syntax) and common Unix shells are available.

In most cases, however, the parameters of these commands are different. For example, in Cmd.exe you could run this command:

```
Dir /s
```

In Windows PowerShell, that same command will generate an error, because */s* is not recognized as a valid parameter.

However, the availability of these familiar command names helps you to start using the shell immediately.

External commands, such as Ipconfig.exe, Pathping.exe, and so on, continue to run exactly as they always have. You can, for example, run this common command from within Windows PowerShell:

```
Ipconfig /all
```

The distinction is that Ipconfig.exe is an external executable, whereas commands such as Dir are internal, or *intrinsic*, commands. This is true in Cmd.exe, as well. Intrinsic commands will usually have a parameter syntax different from what you are used to, and external executables will continue to function as they always have.



Try running some of the command-line commands that you have used in the past. Which ones work the same, and which ones do not?

## Demonstration: Navigating the File System

- Review how to navigate and manipulate the file system within the shell



### Key Points

In this demonstration, you will review how to navigate and manipulate the file system from within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Navigating the File System.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Hierarchical Storage

- The Windows file system is a hierarchical storage system
  - It consists of containers (folders, directories), which can contain leaf elements (files) as well as other containers
- The file system is not the only hierarchical storage system in Windows. Also organized in a hierarchy are:
  - The registry
  - The certificate store
  - Active Directory
- One of PowerShell's key design strategies is to adopt a set of techniques or processes, and then use them for similar tasks
  - The commands to navigate the registry are the same as those used to navigate the file system
  - The commands to navigate Active Directory are the same as those used to navigate the file system

### Key Points

The Windows file system, and the file system of most computers, for that matter, is a *hierarchical* storage system. That is, it consists of containers, called folders or directories, which can contain leaf objects—files—as well as other containers. Folders contain subfolders which contain subfolders, and so on.

Most Windows administrators have memorized the commands needed to manage a hierarchical storage system—Cd, Dir, Copy, Move, and so on. However, the file system is certainly not the only hierarchical storage system in Windows. The registry, the certificate store, Active Directory, and many other storage repositories are also organized in a hierarchy. Windows also contains many flat, or non-hierarchical, forms of storage, such as the operating system's environment variables.

One of the key design strategies of Windows PowerShell is to adopt a set of techniques or processes, and to then use that same set of techniques and processes for any similar task. For example, if both the registry and the file system are hierarchical storage systems, why would you not use the exact same commands to navigate and manipulate both of them?

In other words, because administrators already know a common set of commands to use with one form of hierarchical storage, why not adopt the same set of commands for working with other forms of storage, including flat storage?

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Navigating Other Forms of Storage

- Learn how to navigate other forms of storage using common hierarchical storage commands



### Key Points

In this demonstration, you will learn how to navigate other forms of storage using common hierarchical storage commands.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Navigating other Forms of Storage.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## One Set of Commands for Many Forms of Storage

- The ability to use a single set of commands to navigate many forms of storage is made possible by a PowerShell feature called a *PSDrive provider*
- The PSDrive provider is an adapter that connects to a storage system and presents it to PowerShell
  - This enables common commands, such as cd and dir, to be used against different kinds of storage
  - Commands are passed to the provider of the drive you are accessing, and the provider does whatever is necessary to implement the command
- Several providers ship with PowerShell by default:

The file system	IIS
The registry	SQL Server
The environment variables	Active Directory
The certificate store	

### Key Points

The ability to use a single set of commands to navigate many forms of storage is made possible by a Windows PowerShell feature called a *PSDrive provider*, or simply a *provider*.

A provider is a sort of adapter that connects to a storage system and presents it to Windows PowerShell in the form of a disk drive. When you run commands such as Cd or Dir, those commands are passed to the provider of the drive you are accessing, and the provider does whatever is necessary to implement that command.

Windows PowerShell ships with several providers, including those for:

- The file system
- The registry
- The environment variables
- The certificate store

It also includes providers for the shell's own storage, including storage of functions, aliases, and variables, which you will learn about later in this course.

The shell can be extended to use other providers. Providers already exist for IIS, SQL Server, Active Directory, and many other forms of storage. The providers all make their respective storage repositories appear as disk drives within the shell.

The drives within the shell are officially called PSDrives, or Windows PowerShell Drives. They do not appear outside of the shell in Windows Explorer, and they can have names that are longer than a single letter. For example, the registry drives are HKCU: and HKLM:. You can create new drives by specifying the new drive's name, the provider it will use, and the starting location that the provider will connect to, such as a folder or Universal Naming Convention (UNC) path with the FileSystem provider. Note that the shell

always starts with the same drives when you begin a new shell session; any custom drives you create will be discarded when you close the shell.

MCT USE ONLY. STUDENT USE PROHIBITED

## Managing PSDrives

- The Get-PSDrive cmdlet...
  - Lists all currently-available drives. By default, this includes a FileSystem drive for each logical disk in your computer.
- The New-PSDrive cmdlet...
  - Creates a new drive. You must provide the drive name (which does not include the final colon), the name of the provider to use, and a starting location or path.
    - The type of location or path you provide will depend upon the provider you are using for that drive.
- The Remove-PSDrive cmdlet...
  - Removes an existing drive. You can even remove the default drives like HKCU: or ENV:, although these will be re-created when you open a new shell window or tab.

### Key Points

The shell includes a complete set of commands for managing PSDrives:

- **Get-PSDrive** lists all currently available drives. By default, this includes a FileSystem drive for each logical disk in your computer.
- **New-PSDrive** creates a new drive. You must provide the drive name (which does not include the final colon), the name of the provider to use, and a starting location or path. The type of location or path you provide depends upon the provider you are using for that drive.
- **Remove-PSDrive** removes an existing drive. You can even remove the default drives like HKCU: or ENV:, although these will be recreated when you open a new shell window or tab.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Managing PSDrives

- Learn how to manage PSDrives within the shell



### Key Points

In this demonstration, you will learn how to manage PSDrives within the shell.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Managing PSDrives.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Aliases, Not Commands

- Most of the common commands you've seen in the shell are actually PowerShell aliases
  - Aliases are shorter nicknames for a command
  - Dir is an alias to the Get-ChildItem command
- Cmdlets are built-in commands, which are distinct from external commands such as ipconfig.exe in several ways:
  - Cmdlets use a consistent verb-singularNoun naming convention.
  - Cmdlet verbs are taken from a strictly controlled list of verbs.
  - Cmdlets are written in a .NET Framework language such as Visual Basic or C#, as they are physically packaged in .NET Framework assemblies with a .DLL extension.
  - In reality, the shell does not contain any built-in cmdlets. It loads several snap-ins automatically when you open a new shell window or tab.

### Key Points

Most of the common commands that you have seen in the shell, such as Dir, Cd, Ls, Copy, Rm, and so on, are actually Windows PowerShell *aliases*. Aliases are usually shorter nicknames for a command, and they are designed to make it easier and faster to type common commands, as well as to provide easily-recognizable nicknames for commands you have used in other shells.

For example, *Dir* is actually an alias for the command **Get-ChildItem**. *Ls* is an alias for the same command, as is the alias *Gci*. Aliases do not change the way the command is used, nor do they change its parameters. They are simply a nickname for the command name.

In Windows PowerShell, built-in commands are called *cmdlets* (pronounced, "command-lets"). These are distinct from external executables such as Ipconfig.exe in several ways:

- Cmdlets use a consistent *verb-singularNoun* naming convention. The verbs are taken from a strictly controlled list of approved verbs; you will always use the verb "Get" rather than alternatives like "Retrieve" or "List." This naming convention helps make cmdlet names more consistent, and often easier to guess. Nouns are always singular: "ChildItem" not "ChildItems," or "PSDrive" not "PSDrives." Singular nouns are used even when the cmdlet may return multiple items.
- Cmdlets are written in a .NET Framework language such as Visual Basic or C#, and they are physically packaged in .NET Framework assemblies that have a .dll filename extension. These assemblies are called *snap-ins* or *PSSnapins*. The shell actually does not contain any built-in cmdlets, but it does load several snap-ins automatically when you open a new shell window or tab.

The fact that Dir is an alias helps explain why **Dir /s** doesn't work; you are not running the familiar Dir command, but rather Get-ChildItem cmdlet, for which Dir is an alias.

## Managing Aliases

- You are not limited to using the shell's pre-defined aliases.
  - You can create your own aliases
  - You can import and export aliases to share them with colleagues and coworkers
- Get-Alias displays a list of all defined aliases. You could also run Dir Alias: to see a directory listing of the ALIAS: drive.
- New-Alias creates a new alias. You must provide the alias name as well as the name of the command for which it will be a nickname.
- Del or Rm can be used to delete aliases from the ALIAS: drive.
- Import-Alias and Export-Alias enables you to import and export alias lists from and to a file.

### Key Points

You are not limited to using the aliases that are pre-defined within Windows PowerShell. You can create your own aliases, and you can import and export aliases to share them with colleagues and coworkers. You can even remove aliases—Windows PowerShell stores its aliases in the ALIAS: drive, so deleting an item from that drive removes the alias. Don't worry about accidentally deleting one of the predefined aliases. Like many things in the shell, every time you open a new shell window or tab, you start fresh with the same pre-defined aliases. That also means any custom aliases that you create will be lost when you close your shell window. Exporting the aliases to a file will enable you to more easily import those same aliases again in the future.

**Note:** You can also create an auto-executing *profile* that defines aliases and performs other startup tasks each time you open a new shell window. You will learn about profiles in Module 8 of this course.

There are several commands that you can use to manage aliases:

- **Get-Alias** displays a list of all defined aliases. You could also run Dir Alias: to see a directory listing of the ALIAS: drive.
- **New-Alias** creates a new alias. You must provide the alias name as well as the name of the command for which it will be a nickname.
- **Del** or **Rm** can be used to delete aliases from the ALIAS: drive.
- **Import-Alias** and **Export-Alias** enables you to import and export alias lists from and to a file.

## Asking for Help

- PowerShell at its core is defined for discoverability
    - It includes a built-in help system, aliased to Help
- Help commandName**
- The PowerShell help system uses several parameters to reveal more information about commands
    - -detailed displays more detailed help
    - -examples will display usage examples
    - -full displays everything, including detailed help, per-parameter help, and usage examples
    - -online opens a Web browser and displays the cmdlet help from the Microsoft Web site. This Web site can contain updated or expanded information that has not yet been released in a product update or service pack.

### Key Points

One of the great advantages of a graphical user interface (GUI) is *discoverability*. You have probably used a GUI's discoverability features without even realizing it: Browsing through menus to see what is available, right-clicking things to see what actions are on the context menu, and hovering over items to see what tool tip pops up are all features designed to help you figure out what the GUI can do, without actually making you take any action.

In a command-line shell, these discoverability features are obviously missing. Traditionally, that has made command-line shells harder to learn, because there is no obvious starting place. Windows PowerShell, however, implements several features designed to let you explore and figure out what capabilities are available to you.

One of these features is the built-in help system. Most cmdlets provided by Microsoft include extremely detailed help. Simply run the **Help** command, followed by a cmdlet name or alias, and the shell displays the help for that cmdlet. In cases where you ask for help with an alias, you see the help for the cmdlet, which can help you learn the real name of the cmdlet if all you knew was the alias.



Try to discover the cmdlet name for aliases such as **Type**, **Cat**, and **Del**.

**Note:** If you are familiar with Unix command names, you can run **Man** instead of **Help** to obtain the same results.

The Help command also accepts several parameters that can reveal even more information:

- **-detailed** displays more detailed help.
- **-examples** displays usage examples.
- **-full** displays everything, including detailed help, per-parameter help, and usage examples.

- **-online** opens a Web browser and displays the cmdlet help from the Microsoft Web site. This Web site can contain updated or expanded information that has not yet been released in a product update or service pack.

For example, to see the complete help file for the Get-ChildItem command, you might run:

```
Help gci -full
```

If you learn nothing else from this course, learn to use the shell's help system. Become comfortable with the help system. Use it frequently, both to remind yourself of the correct cmdlet syntax and to learn about new cmdlet features.

**Note:** You should review the help file for every cmdlet you encounter in this course. In most cases, the cmdlets you learn about will have additional capabilities that may be useful to you someday, and simply knowing that additional capabilities exist will help remind you to research them again when the need arises.

You should also get into the habit of using the **-example** parameter with the **Help** command. The usage examples are the best way to discover new cmdlet capabilities, see the correct syntax, and so on. There's no need to waste time digging for examples on the Internet—the examples are built right in, and are available whenever you need them.

**Note:** It can sometimes be annoying to have to look up the help when you are already in the middle of typing a lengthy command. Remember that you can always open another shell window or tab, and run the help there, while continuing to work in your original window or tab.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Asking for Help

- Learn how to use the Windows PowerShell help system



### Key Points

In this demonstration, you will learn how to use the Windows PowerShell help system.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Asking for Help.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Consistency Is Discoverability

- Another of PowerShell's discoverability features is in its consistent cmdlet names
  - Their verb-singular Noun construction enables you to often guess the proper command for accomplishing a task
- The Get-Command cmdlet...
  - Makes it easier to locate a cmdlet with a partial cmdlet name, or a guess
  - Get-Command accepts wildcards, so you can run Get-Command \*proc\* to discover cmdlets that have "proc" as part of their name
  - Get-Command has a -verb parameter, so you can run Get-Command -verb get to see all "Get" cmdlets
  - Get-Command has a -noun parameter, so you can run Get-Command -noun \*event\* to see all cmdlets that have "event" as part of their noun

### Key Points

Another of the Windows PowerShell discoverability features is the consistent cmdlet names. As you become more familiar with the verbs used by the cmdlets, you will start to be able to guess cmdlet names.

For example, if Get-PSDrive displays a list of PSDrives, what cmdlet would display a list of running processes? What about the services on your computer? If you were using Microsoft Exchange Server, what cmdlet might display a list of mailboxes?

If you guessed **Get-Process**, **Get-Service**, and **Get-Mailbox**, you're correct—and you're starting to see why consistent cmdlet naming can be so helpful. After guessing a cmdlet name, you can always use the Help command to see if you guessed correctly.

The Help command also accepts wildcards, meaning you can use less precise guesses about cmdlet names. For example, running this command:

```
Help *Process*
```

displays a list of all help topics that include the word "process." Sometimes, you may run across help topics that do not relate to a specific command, but instead cover an entire concept.



Try running **Help About\*** to see what non-cmdlet help topics are available.

Windows PowerShell also has a cmdlet that works with cmdlets, called **Get-Command** (it has an alias, Gcm). This command can also make it easier to locate cmdlet with a partial cmdlet name—or a guess:

- Get-Command accepts wildcards, so you can run Get-Command \*proc\* to discover cmdlets that have "proc" as part of their name.

- Get-Command has a –verb parameter, so you can run Get-Command –verb get to see all “Get” cmdlets.
- Get-Command has a –noun parameter, so you can run Get-Command –noun \*event\* to see all cmdlets that have “event” as part of their noun.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Discovering Commands

- Learn how to discover cmdlet names using a variety of shell discoverability techniques



### Key Points

In this demonstration, you will learn how to discover cmdlet names using a variety of shell discoverability techniques.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Discovering Commands.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Reading Help and Using Parameters

- PowerShell cmdlets typically support one or more parameters
  - The help file for each cmdlet includes a syntax summary for its associated parameters
  - Some parameters, called switches, don't require a value
    - Recurse**
  - Other parameters require a value, with the help file telling you what kind of value the parameter expects
    - Exclude <string[]>**
  - Even other parameters have required values, which must be provided in order for the cmdlet to function
    - [-Name] <string>**
  - Finally, some parameters are optional
    - [-Include <string[]>]**

### Key Points

You will have noticed from reading the help files that Windows PowerShell cmdlets typically support one or more parameters. The syntax summary of the help files can provide you with several useful clues about how the parameters work.

Some parameters do not require a value. These are called *switches*, and you either specify the parameter or you don't. You can distinguish these parameters in the help because their name is not followed by a value type:

**-Recurse**

Other parameters do require an input value, and the help tells you what kind of value, such as a number or a string, the parameter expects:

**-Exclude <string[]>**

Some parameters are required, and must be provided in order for the cmdlet to function. If you forget to do so, the shell will usually prompt you for the missing information:

```
PS C:\> new-aduser
cmdlet New-ADUser at command pipeline position 1
Supply values for the following parameters:
Name:
```

Other parameters are optional, and these are easy to see in the help because they and their value type are entirely enclosed in square brackets:

**[-Include <string[]>]**

**Note:** The string[] value type, with the opposing square brackets immediately after the value type name, indicates that the parameter can accept more than one value. One way to provide multiple values is to type a list of values, separating each value with a comma.

Many parameters are named, meaning that you must type the parameter name and its value. These parameters can be distinguished by the absence of square brackets around the parameter name, even if the entire parameter is optional:

```
[-Exclude <string[]>]
```

In other words, if you choose to use this parameter, you must provide the parameter name. Parameter names are always preceded by a dash or hyphen, and a space always separates the parameter name from its value:

```
-Exclude *.d11"
```

Other parameters are *positional*. You see this in the help with square brackets around *just* the parameter name:

```
[-Path] <string[]>
```

Positional parameters permit you to type only the value name, provided you type it in the correct position. That is, if –LiteralPath was the first parameter listed in the help, then you could simply provide a string in the first position, without typing the parameter name:

```
Get-ChildItem *.*
```

You can also see the correct position number by reviewing the –full help of a command:

```
-Path <string[]>
    Specifies a path to one or more locati
        Required?          false
        Position?         1
        Default value
        Accept pipeline input?   true (ByV
        Accept wildcard characters? false
```

When you do type a parameter name, you do not need to type all of it. You only need to type enough of the name so that Windows PowerShell can determine exactly which parameter you are referring to. For example, rather than typing

**-computerName**, you could might type **-comp**, or even **-c**, provided the cmdlet has no other parameters whose names started with the letter C.



Review the help for Gsv. What is the full cmdlet name? What parameters are optional? Which ones are positional? Do any parameters accept more than one value? Would -E be a valid shortening of the -Exclude parameter?

## Extending the Shell

- The default cmdlets of the shell are not the only ones available
  - Microsoft and third-party software developers create additional cmdlets and PSDrive providers
- Additional cmdlets are made available through snap-ins and modules
  - Get-PSSnapin
  - Add-PSSnapin
  - Remove-PSSnapin
  - Import-Module
  - Remove-Module
  - Get-Module

**Get-PSSnapin -registered  
Get-Module -list**

### Key Points

The cmdlets available by default within the shell are not the only ones you may have available to you. Microsoft and third-party software developers can create additional cmdlets, as well as additional PSDrive providers, and make them available to you as snap-ins or as *modules*.

Snap-ins are managed with a set of cmdlets using the noun “PSSnapin,” including:

- Get-PSSnapin
- Add-PSSnapin
- Remove-PSSnapin

Snap-ins must typically be installed using an installation package, because they must be registered with the shell. To see a list of registered snap-ins (other than the default ones that ship with the shell), run this command:

```
Get-PSSnapin -registered
```



What other capabilities does Get-PSSnapin have? Try reading the help file to find out.

Modules do not necessarily require an installation, although that can differ depending on exactly what the module does. Windows PowerShell has three main cmdlets for managing modules:

- Import-Module
- Remove-Module
- Get-Module

To see a list of installed modules, run this command:

```
Get-Module -list
```

Windows PowerShell looks for modules in specific folders. To see which folders the shell will search, run this command:

```
Type Env:\PSModulePath
```

You can modify that environment variable to change or add folders where modules are stored.

Note that extensions can sometimes override default shell cmdlets, changing their behavior.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Extending the Shell

- Learn how to extend Windows PowerShell



### Key Points

In this demonstration, you will learn how to extend Windows PowerShell.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Extending the Shell.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Mini-Shells

- Software developers can create mini-shells
  - A mini-shell is essentially a subset of the normal features of PowerShell
  - In a mini-shell, a developer may add new cmdlets, block access to cmdlets, or block access to certain parameters of cmdlets
  - The purpose of mini-shells is to provide a constrained environment that helps ensure technical consistency
  - They help ensure that product-specific extensions don't conflict with other extensions that are already loaded
- SQL Server 2008 ships with a mini-shell
  - You cannot load other extensions within its mini-shell
  - You can load a normal PowerShell window and manually load the SQL Server 2008 extensions

### Key Points

Software developers, especially Microsoft, developers, can create something called a mini-shell. A mini-shell is essentially a subset of Windows PowerShell's normal features. In a mini-shell, the developer may add new cmdlets, but may also block access to some of the shell's default cmdlets. The developer can also block access to particular parameters of a cmdlet.

The main purpose of a mini-shell is to provide a constrained environment to help ensure technical consistency. Consider a product that depends upon Windows PowerShell, such as Microsoft SQL Server®. The SQL Server product team can extend the shell by writing their own modules or snap-ins. However, in the regular shell, their extensions could potentially conflict with other extensions that you, the user, have loaded into your shell. These conflicts can lead to product support difficulties. The mini-shell helps address this: If you have a problem using one of the SQL Server cmdlets, a product support representative might ask you to try the same cmdlet in the mini-shell, which might block the capability to load your own extensions. The product support team knows how the mini-shell is configured, and knows that your ability to modify that configuration is limited. If you are able to duplicate the problem in the mini-shell, then they know the problem is not due to a conflict with other extensions.

A mini-shell does not stop you from using a product's shell extensions with other extensions. For example, SQL Server 2008 shipped with a mini-shell. Within that mini-shell, you could not load other extensions. However, you could certainly open a normal Windows PowerShell window and manually load the SQL Server 2008 extensions with whatever other extensions you wished. A product's mini-shell is not necessarily the only means of using that product's shell extensions.

## Progress Check

- You have at this point learned most of the major skills you need to use PowerShell effectively!
- You know how to:
  - Add shell extensions
  - Get a list of available cmdlets
  - Locate, display, and work with help
  - Run cmdlets and see their output
- Everything going forward will build upon these simple foundation skills
  - Using these skills, you can teach yourself to use any new PowerShell feature or extension
  - The rest of this course will focus on teaching different cmdlets and the specialized skills for using them effectively

## Key Points

Believe it or not, you have at this point learned most of the major skills needed to use Windows PowerShell effectively. You know how to add shell extensions, and you know how to get a list of available cmdlets and help topics. You know how to display help files, which show you the proper syntax and usage examples for running cmdlets. You know how to run cmdlets and see their output.

While there is certainly a lot to learn, everything going forward will build upon these simple foundation skills. Using these basic skills, you should be able to teach yourself to use any new Windows PowerShell feature or extension that you may come across. Much of the rest of this course will focus on teaching you different cmdlets, and teaching you specialized techniques for using those cmdlets effectively.

## Cmdlet Output

- PowerShell includes a series of built-in defaults that define how cmdlet output is displayed
- Three main cmdlets are useful in customizing cmdlet output:
  - Format-Table, which uses the alias `Ft`
  - Format-List, which uses the alias `Fl`
  - Format-Wide, which uses the alias `Fw`

**Get-Service | Format-List  
Get-Process | Fw**

### Key Points

Windows PowerShell has a series of built-in defaults that define how the output for cmdlets is displayed. You will learn more about these defaults and how to manipulate them later in this course, but for now you may find it useful to perform some basic customizations to cmdlet output.

There are three main cmdlets that can be useful in customizing cmdlet output:

- Format-Table, which uses the alias `Ft`
- Format-List, which uses the alias `Fl`
- Format-Wide, which uses the alias `Fw`

These cmdlets change a cmdlet's output to a table, a list, or a multi-column wide display, respectively. To use these cmdlets, simply *pipe* the output of a cmdlet to one of these cmdlets or their aliases:

**Get-Service | Format-List  
Get-Process | Fw**



What other formatting cmdlets are available in the shell? Use Help or Get-Command to find out.

## Piping

- The act of directing the output from one cmdlet to the input of another is called piping

- You have seen piping before:

**dir | more**

- Piping is used extensively in PowerShell

- It is not uncommon to string together a sequence of a half-dozen cmdlets, all connected with pipes
- Data flows from one cmdlet to another, gradually refining and manipulating that data into what is needed at the time
- PowerShell cmdlets are designed to work this way
- In fact, Piping is so powerful and important that the entire next lesson is dedicated to its topic

### Key Points

The act of directing the output from one cmdlet to the input of another cmdlet is called *piping*. You may have used piping in other shells. For example, this is a common command in Cmd.exe:

Dir | More

Here, the output of the Dir command is directed to the More command, which creates a paged display of the output.

Piping is used extensively in Windows PowerShell. It is not uncommon to string together a sequence of a half-dozen cmdlets, all connected with pipes. Data flows from one cmdlet to another, gradually refining and manipulating that data into whatever it is you need at the time. Windows PowerShell cmdlets are designed to work in this fashion, more so than in traditional shells in either Windows or Unix. Piping can enable you to accomplish very complex and powerful tasks without having to write complicated scripts or programs.

Piping is such a powerful and important concept that the entire next lesson of this course is dedicated to the topic.

## Demonstration: Customizing Cmdlet Output

- Learn how to change the format of cmdlet output



### Key Points

In this demonstration, you will learn how to change the format of cmdlet output.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Customizing Cmdlet Output.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab A: Using Windows PowerShell As an Interactive Command-Line Shell

- Exercise 1: Searching for Text Files
- Exercise 2: Browsing the Registry
- Exercise 3: Discovering Additional Commands and View Help
- Exercise 4: Adding Additional Commands to Your Session
- Exercise 5: Learning How to Format Output

Logon information

• Virtual machine	LON-DC1
• Logon user name	Contoso\Administrator
• Password	Pa\$\$w0rd

**Estimated time: 50 minutes**

**Estimated time: 50 minutes**

You work as a systems administrator. You have installed Windows Server 2008 R2 and need to learn how to automate some tasks. You heard that PowerShell was great for this sort of thing and need to start learning how to use it. You have used command-line utilities in the past and want to get started working with PowerShell as quickly as possible. You're also looking for tips on how you can discover things in PowerShell on your own.

### Lab Setup

For this lab, you will be browsing the file system and Registry and learning how to use PowerShell on a domain controller using a domain administrator credential. Before you begin the lab you must:

1. Start the **LON-DC1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open a Windows PowerShell session as **Administrator**.
3. Open Windows PowerShell and execute the following command:

"These are my notes" | Out-File C:\users\Administrator\Documents\notes.txt

## Exercise 1: Searching for Text Files

### Scenario

You are working with another administrator on a Windows Server 2008 R2 domain controller. The other administrator had been documenting his work in a text file in a user profile, but you're not sure which profile has the file. You are taking a colleague's advice and starting to use PowerShell to perform tasks you would previously have done with cmd.exe to find the file.

The main tasks for this exercise are as follows:

1. Browse the local file system using familiar command prompt and/or UNIX commands.
2. View the entire contents of one user's profile folder.
3. View a list of all text files in all users' document folders.

► **Task 1: Browse the local file system using familiar command prompt and/or UNIX commands**

- Open **Windows PowerShell** and set the current location to **C:\**.
- Show the contents of the **C:\users** folder from your current location.

**Hint:** What command would you use to see a *directory listing*? Read the help for that command and see if it has a parameter that lets you specify the *path* of the folder that you want to see a directory listing for.

- Show the contents of the **C:\users** folder from within that folder.

**Hint:** What command would you use to *change directories*? Use that command to change to the **C:\Users** folder.

- Create a folder called **Test** at the root of the **C** drive.

**Hint:** What command would you use to *make a directory*? Use that command to create the new folder.

- Remove the **C:\Test** folder.

► **Task 2: View the entire contents of a folder**

- Get the entire list of all files in all folders and subfolders of the user profile folder for the **Administrator** account.

**Hint:** What command would you use to see a *directory listing*? Does that command have a parameter that would enable to you specify the *path* of the directory you want to see a listing for? Is there a parameter that would *recurse* subdirectories?

- Repeat the last activity using the shortest parameter names possible.

**Hint:** You only need to type enough of a parameter name so that Windows PowerShell can uniquely identify the parameter you are referring to. “-comp” is often easier to type than “-computername,” for example.

► **Task 3: View a list of all text files in all users’ document folders**

- Get the list of all text files in all subfolders of all user profile folders.

**Hint:** You know the command that will list files and folders. Does that command support a parameter that would enable you to *include* only certain types of files, such as \*.txt?

- Open the notes.txt file in the **Administrator** account’s documents folder in Notepad.

**Results:** After this exercise, you should have successfully navigated the file system, discovered the notes.txt file in the Administrator’s user profile folder, and viewed the contents of that file in notepad.

## Exercise 2: Browsing the Registry

### Scenario

You are working on a shared computer and want to identify differences in startup programs between different accounts. You must show the different startup programs for the current user profile and built-in accounts on screen, but you don't have to actually do the comparison.

The main tasks for this exercise are as follows:

1. View cmdlet help.
2. Navigate the Registry.
3. Create a new PSDrive.

► **Task 1: View cmdlet help**

- View basic help information for the **Get-PSDrive** cmdlet using Get-Help.
- View full help information for the **Get-PSDrive** cmdlet using Get-Help with the **-Full** parameter.

► **Task 2: Navigate the registry**

- Show a list of all available PSDrives using the **Get-PSDrive** cmdlet.
- Change the current location to **HKLM:\Software\Microsoft**.
- Show a list of all available PSDrives for the **Registry** provider.
- Change the current location to the **HKEY\_CURRENT\_USER** registry hive and list the contents at the root of that drive.

► **Task 3: Create a new PSDrive**

- Create a new PSDrive using the **New-PSDrive** cmdlet. This drive should be named "**HKU**" and it should have the **HKEY\_USERS Registry** hive as the root.

**Hint:** Read the help to discover the three pieces of information that the New-PSDrive cmdlet needs in order to create a new drive. The PSProvider for the Registry is called "Registry."

- View a list of all Registry PSDrives including the HKU drive you just added using the **Get-PSDrive** cmdlet.

## Exercise 3: Discovering Additional Commands and Viewing Help

### Scenario

To properly use PowerShell, you need to know how to identify commands, discover new commands, and get help information to learn how to use commands.

The main tasks for this exercise are as follows:

1. Discover commands using aliases and aliases using commands.
2. Learn how to use the help system using Get-Help.
3. Discover new commands with Get-Command.
4. Get current, online help for a command.

► **Task 1: Discover commands using aliases and aliases using commands**

- Find the command associated with the alias "dir" using **Get-Alias**.
- Find other aliases for the same command using **Get-Alias** with the **Definition** parameter.

► **Task 2: Learn how to use the help system using Get-Help**

- View basic help information for the **Get-Help** cmdlet using that cmdlet itself.
- View the examples for **Get-Help** using the help proxy command.
- Show the full help for **Get-Help** using the help proxy command.

► **Task 3: Discover new commands with Get-Command**

- Find all commands with "Service" in their name using **Get-Command** and wildcards.
- View basic help information for **Get-Command** using the help command.
- Find all commands of type cmdlet with "Service" in their name using **Get-Command**, wildcards and the  **CommandType** parameter.
- Find all commands with the "Service" noun using **Get-Command** with the **Noun** parameter.

► **Task 4: Get current, online help for a command**

- Get the current, online help for the **Get-Service** cmdlet using the **Get-Help** cmdlet.

**Results:** After this exercise, you should know how to discover commands using aliases or wildcards, how to look up different parts of help information for a command, how to use help information to learn how to use a command, and how to get current help online.

## Exercise 4: Adding Additional Commands to Your Session

### Scenario

When working with some Windows components and server products, you must load modules or snap-ins to be able to access the commands they provide.

The main tasks for this exercise are as follows:

1. Find all “Module” commands.
2. List all modules that are available.
3. Load the ServerManager module into the current session.
4. View all commands included in the ServerManager module.

► **Task 1: Find all “Module” commands**

- Show all “**Module**” commands using **Get-Command** so you know what is used to manage modules.

► **Task 2: List all modules that are available**

- List all modules that are available on the current system using **Get-Module**.

**Hint:** You can also get a directory listing of the \$pshome/modules folder.

► **Task 3: Load the ServerManager module into the current session**

- View basic help documentation and examples for **Import-Module** so that you know what modules are and how to load them.
- Load the **ServerManager** module into the current session.

► **Task 4: View all commands included in the ServerManager module**

- Show a list of all commands that are part of the **ServerManager** module using **Get-Command**.
- Get a list of the Windows roles and features that are installed or available on the current system using **Get-WindowsFeature**.

**Results:** After this exercise, you should be able to view and load modules into PowerShell and show what commands they contain.

## Exercise 5: Learning How to Format Output

### Scenario

When viewing data in PowerShell, you can format the output to get the results you need.

The main tasks for this exercise are as follows:

1. View the default table format for a command.
2. View the default list and wide formats for a command.

► **Task 1: View the default table format for a command**

- View a list of all processes whose names start with “w” using the **Get-Process** command.
- Show the default table view for process objects using the same command but passing the results to **Format-Table**.

► **Task 2: View the default list and wide formats for a command**

- Using the same **Get-Process** command, show the default list view for process objects by passing the results to **Format-List**.
- Using the same **Get-Process** command, show the default wide view for process objects by passing the results to **Format-Wide**.

**Results:** After this exercise, you should be able to show PowerShell output in table, list and wide formats using default views.

## Lab Review

1. What is the name of the cmdlet with the alias dir?
2. How many Registry PSDrives are available in PowerShell by default?
3. How many commands are there with the noun Service?
4. What command is used to load modules into your current session?
5. How do you format PowerShell output in a list format?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 3

# Using the Windows PowerShell Pipeline

- Explain how the Windows PowerShell pipeline works
- Explain the purpose of attributes and actions
- View the attributes and actions of a given element
- Use the pipeline to pass elements from one cmdlet to another
- Explain how Windows PowerShell takes cmdlets' element output and creates a textual display



Windows PowerShell cmdlets execute in a *pipeline*, which is what enables data to pass from one cmdlet to another. The nature of the pipeline, as well as the nature of the shell's cmdlet output, is what makes Windows PowerShell a unique and powerful tool for accomplishing administrative tasks without having to write complex scripts or programs.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain how the Windows PowerShell pipeline works.
- Explain the purpose of attributes and actions.
- View the attributes and actions of a given operating system element.
- Use the pipeline to pass elements from one cmdlet to another.
- Explain how Windows PowerShell takes cmdlets output and creates a textual display.

## The Pipeline

- All Shell commands execute in something called a pipeline
  - You can imagine this to be a literal pipe through which information can flow
  - Each cmdlet is a “pumping station”, and the pipeline connects each pumping station to the next
  - Each station can take information out of the pipeline, or put information into the pipeline
  - Cmdlets execute in order from left to right, with the pipe character (“|”) providing the delimiter between cmdlets



### Key Points

All Shell cmdlets execute in something called a *pipeline*. You can imagine this to be a literal pipe through which information can flow. Each cmdlet is like a pumping station, and the pipeline connects each pumping station to the next. Each station can take information out of the pipeline, or put information into the pipeline, before allowing the information to proceed down the pipeline to the next station.

Cmdlets execute in order from left to right, and the pipe character (“|”) provides a delimiter between each cmdlet.

You have already seen examples like this one:

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

This is perhaps the simplest possible use of the pipeline. Much more complex and powerful commands can be created. For example, later in this course you will learn to write a command string like this one:

```
Import-Csv c:\new_users.csv | New-ADUser -passThru | Enable-ADAccount
```

This simple three-command string could create hundreds of new Active Directory users and enable the user accounts. In older administrative technologies, this task might have required dozens of lines of scripting language programming; with Windows PowerShell’s task-focused cmdlets and data-passing pipeline, this task can be accomplished using exactly the syntax you see here (provided that a few prerequisites are met, which you will learn about later).

## The Problem with Text

- PowerShell is not the first shell to use a pipeline
  - Cmd.exe and Unix shells have used pipelines in the past
  - The other shells, however, piped text from one command to another
  - Piping text requires that text to be correctly formed as it passes from command to command
  - The following command won't work in cmd.exe:  
**plist | kill**
- Other shells required complex text manipulation (a la grep, sed, awk) to prepare one command's output text for piping into another command's input parameters
  - This is not efficient

### Key Points

Windows PowerShell is certainly not the first shell to use a pipeline. Unix shells have piped data from one command to another for decades, and Windows' own Cmd.exe used the pipe character in much the same way.

The problem with these older shells, however, is that they piped *text* from one command to another. For example, suppose a command named Plist produced a text list of running processes, which looked something like this:

Pid	Name	Image
---	---	---
092	Notepad	notepad.exe
098	Windows Paint	mspaint.exe
112	Calculator	calc.exe
164	Windows PowerShell	powersehll.exe

Suppose your goal was to end all instances of Windows Notepad. You might have a command named Kill, which accepts a process ID (PID) and terminates that process. But you can't just run this command:

**Plist | Kill**

First, you have to figure out which lines of the Plist output pertain to an instance of Notepad; then you have to reduce that line to just the PID that Kill requires.

In Unix operating systems, all of this is accomplished using text-parsing utilities such as Sed, Grep, and Awk. For example, this kind of command is common in Unix:

**ps auxwww | grep httpd**

The problem with this approach is that you spend more time parsing and manipulating text than actually accomplishing whatever task you set out to do. Scripting languages like Perl, in fact, became popular

almost entirely because of their powerful text-parsing capabilities. Administrators using shells like these needed to be expert text manipulators—although text manipulation technically has nothing to do with the administrative tasks at hand.

MCT USE ONLY. STUDENT USE PROHIBITED

## Cmdlets Do Not Produce Text

- PowerShell discards text output in favor of object output
  - A PowerShell cmdlet always produces objects, not text
- Objects are self-contained, functioning pieces of the OS, a server technology, or feature
  - A process is an object
  - A service is an object
  - A file or folder is an object
  - A user account, or group, or OU, or domain is an object.
- Objects have properties, which describe attributes of the object

Pid	Name	Image
092	Notepad	notepad.exe
098	Windows Paint	mspaint.exe
112	Calculator	calc.exe
164	Windows PowerShell	powershell.exe

### Key Points

Windows PowerShell discards text output in favor of another kind of output. A Windows PowerShell cmdlet always produces elements that represent operating system components, rather than text.

What is an *element*? Essentially, an element is a self-contained data structure that describes some functioning piece of the operating system, or a server technology. A process, for example, is a functioning piece of the operating system. A service is a piece of the operating system. A file or folder is a piece of the operating system. A user account is an element of the operating system, as is a group or an organizational unit or the domain itself.

You can think of elements as a kind of data structure that Windows PowerShell can work with very quickly and easily. Rather than having to tell the shell to "skip over ten characters in each row to find the column that contains the process name," you can simply tell it to directly access the Name attribute, because the data structures used in the pipeline are designed in a way to make the attributes easily-accessible.

## Terminology

- Windows uses terminology that sometimes seems related to software development
  - Keep in mind that these are just words
  - Using them does not imply that you need to become a programmer in order to use PowerShell
- For example:
  - You have seen the word *element* to refer to things that a cmdlet places in the pipeline
  - The more formal word for elements is *objects*
  - These objects have *attributes*
  - The formal term for attributes is *properties*

**Don't let PowerShell's terminology turn you off!**

### Key Points

Windows PowerShell uses terminology that sometimes seems related to software development. Keep in mind that these are just words; using them does not imply that you need to become a programmer in order to use Windows PowerShell.

For example, in this module you have seen the word *element* used to refer to the things that a cmdlet places into the pipeline. For example, the Get-Process cmdlet places process elements into the pipeline. The more formal word for elements is *objects*, meaning you would say that the Get-Process cmdlet places process *objects* into the pipeline.

These objects have attributes. For a process, that might include things like its name, ID, memory utilization, and so forth. The formal term for these attributes is *properties*, meaning you would say that the process object has a name property, an ID property, and so on.

## Properties, or Attributes

- Objects typically have one or more properties
  - These properties describe attributes of the object
  - Some shells might represent these attributes in a list or table

Pid	Name	Image
092	Notepad	notepad.exe
098	Windows Paint	mspaint.exe
112	Calculator	calc.exe
164	Windows PowerShell	powershell.exe

- However, as you've already learned, having to manipulate this text is overly time-consuming
  - The data structure of an object, however, makes it easy to access properties of an object

**This course will sometimes use the word properties, and other times use the word attributes**

**The actual words aren't important**

## Key Points

Objects typically have one or more *properties*, which describe various attributes of the object. Other shells might represent these attributes in a list or a table. For example, a process object could be represented in text like this:

Pid	Name	Image
092	Notepad	notepad.exe
098	Windows Paint	mspaint.exe
112	Calculator	calc.exe
164	Windows PowerShell	powershell.exe

However, as you have already learned, having to manipulate that text to do something with it is overly time-consuming. The data structure of an object, on the other hand, makes it very easy to access just a single property of an object, or to access a set of properties.

This course will sometimes use the word *properties*, and other times use the word *attributes*, to help remind you that the actual *word* is not important. What is important is understanding that Windows PowerShell makes it easy to access this information in the pipeline.

## Discussion: Object Attributes

- What are some of the attributes of a Windows service?
- What are some of the attributes of an Active Directory user account?



### Key Points

What are some of the attributes of a Windows service?

- Name
- Status (running or stopped)
- Logon account
- Logon password
- Start mode
- Dependencies

What are some of the attributes of an Active Directory user account?

- Name
- Department
- Title
- Phone number
- Address
- City
- Surname
- Given name

An *object* is just a structured way of storing this information, so that individual attributes can be retrieved and referenced without having to parse a text list or table.

**Note:** You can also think of objects as a kind of mini-database, where each object property is a column in the database. The database engine—in this case, Windows PowerShell—can quickly look up a single column's information and present it to you.

MCT USE ONLY. STUDENT USE PROHIBITED

## Object Properties

- PowerShell cmdlets produce objects
  - PowerShell cmdlets also accept objects as input
  - Specific parameters of cmdlets can be designed to look for specific properties on incoming objects
- For example, the Get-Service cmdlet contains a parameter called `-computerName`

**Get-Service -computerName SEA-SRV2**

- Data for parameter could also be fed from another cmdlet's output, without need for text manipulation



### Key Points

Because properties are individually accessible without having to parse text, they provide faster and easier access to individual pieces of information.

Windows PowerShell cmdlets all produce objects, and they can also accept objects as input. Specific parameters of a cmdlet can be designed to look for specific properties on incoming objects, and if those properties are found, the cmdlet can use the information in the property as input to the parameter.

For example, suppose the Get-Service cmdlet has a parameter named `-computerName`. That parameter would specify the name of a remote computer that you wanted to get services from:

**Get-Service -computerName SEA-SRV2**

That parameter could also be designed by the developer of the cmdlet, to look for a "ComputerName" property on incoming objects. Suppose you had a cmdlet named Get-ComputerInventory that retrieved a list of computers from a configuration database. Each computer in the database would be output as an object, and each object might have several properties, including a "ComputerName" property. If that were the case, you could run a command like this to retrieve services from all of the computers in inventory:

**Get-ComputerInventory | Get-Service**

There's no need to produce the inventory in a textual table, parse out the "Computer Name" column, and so forth, because you're not working with text. Windows PowerShell pipes the objects from the first cmdlet to the second, and makes the connection between the objects' properties and the second cmdlet's parameters.

**Note:** There is no Get-ComputerInventory cmdlet built into Windows PowerShell, but the Get-Service cmdlet operates exactly as described here. You'll see examples of this usage later in the course.

MCT USE ONLY. STUDENT USE PROHIBITED

## Finding Properties

- The Get-Member cmdlet...
  - Will show you the members of an object which has been passed into it
  - Shows the type name, the formal and unique name by which an object is known
  - Is a cmdlet you will find yourself using often, to determine the contents and characteristics of objects

```
Get-EventLog Security -newest 10 | Get-Member  
Get-Process | Format-List *  
Get-Service | Get-Member | Out-GridView
```

### Key Points

Another discoverability feature in Windows PowerShell is the **Get-Member** cmdlet. If you pipe objects to it, it will show you their *members*, which includes the properties of the objects.

**Note:** You should get so accustomed to using Get-Member that you wish for an alias to make typing easier. **Gm** is the alias for this cmdlet.

For example:

```
Get-EventLog Security -newest 10 | Get-Member
```

You can also pipe objects to **Format-List \*** to see a list of the objects that includes every single property. For example:

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

Another good way to review object properties is to use Get-Member in conjunction with the Out-GridView cmdlet:

```
Get-Service | Get-Member | Out-GridView
```

**Note:** The Out-GridView cmdlet requires version 3.5, Service Pack 1, of Microsoft .NET Framework.

In addition to showing the objects' properties, Get-Member shows the objects' *type name*, which is the formal, unique name by which that type of object is known.

Being able to obtain a list of object properties is an extremely important skill in Windows PowerShell, and using Get-Member or Format-List is much faster than searching on the Internet. However, if you do need to learn more about a particular type of object, its type name is a good search term to start with.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Properties

- Learn how to find and use object properties in the shell



### Key Points

In this demonstration, you will learn how to find and use object properties in the shell.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Working with Properties.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Common Parameters

- Every cmdlet includes <CommonParameters> along with its other parameters

- These are a short list of parameters that the shell automatically adds to every object
- To view common parameters, use the command:

**help about\_common\_parameters**

- Two useful common parameters are:

- **-confirm** prompts you to confirm the operation on each object that is processed
- **-whatif** does not actually perform the operation, but instead creates a list of what operations would have been performed

### Key Points

You may have noticed that the help for every cmdlet lists <CommonParameters> along with the cmdlet's other parameters.

The Common Parameters are a short set of parameters that the shell automatically adds to every cmdlet. To learn more about them, run this command:

**Help about\_common\_parameters**

Two other useful parameters you may see in the help are **-whatif** and **-confirm**. These are not supported by every cmdlet, but they are supported by most cmdlets that may modify the system in some way.

- **-confirm** prompts you to confirm the operation on each object that is processed
- **-whatif** does not actually perform the operation, but instead creates a list of what operations would have been performed

These can be very useful for testing complicated commands. For example, consider (but please do not run) this command:

**Get-process | Stop-Process**

What will this command do? One easy way to find out is to add **-whatif**:

**Get-process | Stop-Process -whatif**

You will see output similar to the following:

```
What if: Performing operation "Stop-Process" on Target "conhost (2440)".  
What if: Performing operation "Stop-Process" on Target "conhost (2676)".  
What if: Performing operation "Stop-Process" on Target "conhost (2884)".  
What if: Performing operation "Stop-Process" on Target "csrss (316)".
```

What if: Performing operation "Stop-Process" on Target "csrss (368)".  
What if: Performing operation "Stop-Process" on Target "dfsrs (1332)".

This lets you know exactly what would have happened, with less risk of damaging the system.

MCT USE ONLY. STUDENT USE PROHIBITED

## Redirecting Output

- The shell provides several Out- cmdlets, which are used to redirect command output to a different location
  - Out-GridView
  - Out-Printer
  - Out-File
  - Out-Host
  - Out-Null
- Refer to these cmdlets' help files for parameters that further specify output characteristics

### Key Points

The shell provides several Out- cmdlets in addition to the Out-GridView cmdlet you have already seen. Each Out- cmdlet is designed to redirect output to a different location:

- Out-Printer
- Out-File
- Out-Host

Out-Host directs output to the console window, which is the default action that you have already seen. Out-File is extremely useful, and when given a filename can direct output to a file. The output will retain whatever formatting it already had, meaning you can precede an Out- cmdlet with one of the Format- cmdlets:

```
Get-EventLog Application -newest 100 | Format-List * | Out-File events.txt
```

Remember to read the help files! Out-File, for example, has numerous parameters that let you specify character encoding, the desired output width of the text file, and so on.

Out-Null is another option, and has a special purpose. Sometimes, cmdlets return output that you do not want to see. To suppress this output, simply pipe it to Out-Null:

```
Get-Service | Out-Null
```

You can also use the older > and >> syntax to redirect output to a file. These use Out-File behind the scenes, and provide a simpler way of sending output to a text file, although you cannot use any of Out- File's parameters to customize or control the output.

```
Get-Process > procs.txt
```

**Note:** If you're not familiar with this syntax, > redirects output to a new file, overwriting any existing file with the same filename (which is the default behavior for Out-File as well). >> appends the output to an existing file, which is the same as using the –append parameter of Out-File.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Redirecting Output

- Learn how to use output redirection



### Key Points

In this demonstration, you will learn how to use output redirection.

#### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration steps in virtual machine **LON-DC1** located in **E:\Mod01\Democode\Redirecting Output.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## The Power of a Pipeline

- PowerShell passes objects in the pipeline until the very end of the pipeline
  - At the end of the pipeline, whatever objects remain are passed to a special cmdlet called Out-Default
  - This passing to Out-Default is automatic
  - Out-Default passes objects to Out-Host by default , which produces text that you read on the screen
- Because objects are not converted to text until the last step, you can continue to use their properties until the very end



### Key Points

Windows PowerShell passes objects in the pipeline until the very end of the pipeline. At the end of the pipeline, whatever objects remain are passed to a special cmdlet called Out-Default. You don't need to specify this; it's built into the end of the pipeline. Its job is to take whatever objects are left in the pipeline, pass them to the console host cmdlet (Out-Host), and produce the text output you see on the screen. Module 3 of this course will cover that process in more detail.

Because objects are not converted to text until the end of the pipeline, you can continue to access their properties until the very end. That means complex command strings can be created without having to parse text.

There are a couple of exceptions to this general rule:

- Format- cmdlets produce objects, but they are a special type of object that is only understood by an Out- cmdlet. That means a Format- cmdlet needs to be either the last cmdlet on the pipeline, or it needs to be the next-to-the-last cmdlet immediately preceding an Out-cmdlet.
- Most Out- cmdlets do not produce output of any kind; they direct output, as text, to a device, such as a file, a printer, or the console window. If you use an Out- cmdlet, it should usually be the last cmdlet on the pipeline.

Much of Windows PowerShell power and flexibility comes from the ability to pass objects from one cmdlet to another in the pipeline. Because these command strings—even very long ones—occupy a single logical line of text within the shell, they are often referred to as *one-liners*. In many cases, a one-liner can accomplish the same thing that would require many lines of scripting language code in an older technology such as Visual Basic Scripting Edition (VBScript).

For example, this one-liner, which uses techniques we will cover later in this course, connects to all of the computer names listed in c:\computers.txt and produces a report of event log sizes.

```
Get-Content c:\computers.txt | where {($_.Trim()).Length -gt 0} | foreach {
```

```
Get-WmiObject Win32_NTEventLogFile -computer $_.Trim() ` 
-filter "NumberOfRecords > 0" | Select-Object ` 
@{Name="Computername";Expression={$_.CSName}},LogFileName,NumberOfRecords,` 
@{Name="Size(KB)";Expression={$_.FileSize/1kb}},` 
@{Name="MaxSize(KB)";Expression= "{$_.MaxFileSize/1KB) -as [int]}},` 
@{name="PercentUsed";Expression={"{0:P2}" -f ($_.filesize/$_.maxFileSize)}}` 
} | Sort Computername | Format-Table -GroupBy Computername ` 
-property LogFileName,NumberOfRecords,*Size*,PercentUsed
```

Obviously this is a complex command, but it illustrates the power and flexibility of the shell's pipeline.

## Caution: Never Make Assumptions

- Don't make assumptions about property names or the values they contain
  - For example, don't assume that a "DriveLetter" property exists on an object when the actual property is in fact "DeviceID"
- Always use Get-Member to verify property names, and pipe objects to Format-List \* to verify the contents of those properties
  - This process takes very little time to verify, but it will save you a great deal of time and frustration

### Key Points

As you begin working with Windows PowerShell, a major opportunity for error and frustration is when you make assumptions about property names or the values they contain. For example, if you assume that a "DriveLetter" property exists on an object, when in fact the property name is "DeviceID," this can take several minutes to figure out. Creating a one-liner or script that assumes a value of "FixedDisk" in the "DriveType" property will create problems if the property actually contains the value "3."

The lesson here is to never make assumptions. Always use Get-Member to verify property names, and pipe objects to **Format-List \*** to verify the contents of those properties. It takes very little time to verify, and you will save yourself a great deal of time and frustration by doing so.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Using the Windows PowerShell Pipeline

- Exercise 1: Stopping and Restarting a Windows Service
- Exercise 2: Exploring Objects Returned by PowerShell Commands
- Exercise 3: Processing PowerShell Output

Logon information

Virtual machine	LON-DC1	LON-CLI1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are the server administrator in an enterprise. You have a computer on which you want to make administrative changes, such as working with Services. Working with those services using the GUI adds administrative overhead, so you wish to use Windows PowerShell to accomplish the necessary tasks. To better understand how Windows PowerShell interacts with those services, you also need to explore the members, properties, and output options for the necessary commands.

#### Lab Setup

For this lab, you will be browsing the stopping and restarting services and viewing event log entries on a domain controller using a domain administrator credential. Before you begin the lab, you must:

1. Start the **LON-DC1** virtual machine. You do not need to log on, but wait until the boot process is complete.
2. Start the **LON-CLI1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Stopping and Restarting a Windows Service

### Scenario

You have used PowerShell to view processes and services on various computers and now you want to perform some tasks to change the configuration on a computer.

The main tasks for this exercise are as follows:

1. View the Windows Update service details.
2. Determine what would happen if you stopped the Windows Update service.
3. Stop the Windows Update service with confirmation.
4. Restart the Windows Update service.

► **Task 1: View the Windows Update service**

- Open Windows PowerShell.
- Use the **Get-Service** cmdlet to view the Windows Update service.

**Hint:** Do not run Get-Service by itself. Rather, include a parameter so that only the Windows Update service is returned.

► **Task 2: Determine what would happen if you stopped the service**

- Use a pipeline to pass the results of your last **Get-Service** call to the **Stop-Service** cmdlet with the **WhatIf** parameter to see what would happen.

► **Task 3: Stop the Windows Update service with confirmation**

- Use a pipeline to pass the results of your last **Get-Service** call to the **Stop-Service** cmdlet with the **Confirm** parameter. When you are prompted press **Y** to stop the service.

► **Task 4: Restart the Windows Update service**

- Use a pipeline to pass the results of your last **Get-Service** call to the **Start-Service** cmdlet to restart the service.

**Results:** After this exercise, you should have successfully stopped and restarted the Windows Update service and learned about how the **WhatIf** and **Confirm** common parameters can prevent accidental changes.

## Exercise 2: Exploring Objects Returned by PowerShell Commands

### Scenario

You are using PowerShell to manage files and services and you know that everything returned by the commands you use are objects. You want to learn more about the properties and methods on those objects so that you can use them directly in your work to generate the reports you need.

The main tasks for this exercise are as follows:

1. View cmdlet help.
2. Get visible members for Service objects.
3. Get properties for Service objects.
4. Get all members for Service objects.
5. Get base and adapted or extended members for Service objects.
6. Find properties by wildcard search and show them in a table.

► **Task 1: View cmdlet help**

- View full help information for the **Get-Member** cmdlet.

► **Task 2: Get visible members for Service objects**

- Use **Get-Member** in a pipeline to show all visible members for **Service** objects.

► **Task 3: Get properties for Service objects**

- Use **Get-Member** in a pipeline to show all properties that are available for **Service** objects.

► **Task 4: Get all members for Service objects**

- Use **Get-Member** in a pipeline to show all members that are available for **Service** objects.

► **Task 5: Get base and extended or adapted members for Service objects**

- Use **Get-Member** in a pipeline to show a view containing all of the original (without extension or adaptation) members that are available in **Service** objects.
- Use **Get-Member** in a pipeline to show a view containing all of the extended and adapted members that are available in **Service** objects.

► **Task 6: Find properties by wildcard search and show them in a table**

- Get a list of all files and folders (hidden and visible) at the root of the C: drive.
- Show all properties on files and folders that contain the string “**Time**”.
- Generate a table showing all files and folders at the root of the **C:** drive with their name, creation time, last access time and last write time.

**Hint:** The Get-Member cmdlet supports several parameters that let you customize the information it provides.

**Results:** After this exercise, you should be able to use Get-Member to discover properties and methods on data returned from PowerShell cmdlets and use that information to format tables with specific columns.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Processing PowerShell Output

### Scenario

You have used PowerShell to generate output in the PowerShell console, and now you would like to save the output to a file or work with the output in a Windows dialog so that you can sort and filter the output to get the results you need.

The main tasks for this exercise are as follows:

1. List all commands that are designed to process output.
2. Show the default output for a command.
3. Send command output to a file.
4. Send command output to a grid and sort and filter the results.

► **Task 1: List all commands that are designed to process output**

- Show all commands with the verb “**Out**” in their name.

**Hint:** The Get-Command cmdlet supports parameters that let you customize its behavior. For example, one parameter enables you to only retrieve the cmdlets that have a particular verb.

► **Task 2: Show the default output for a command**

- Review the help information for the **Get-EventLog** cmdlet including the parameters and the examples so that you understand how it works.
- Show the 100 most recent events from the **System** event log of type warning or error in the default output.

**Note:** The Get-EventLog cmdlet includes several parameters that can help filter the type of information that the cmdlet returns.

► **Task 3: Send command output to a file**

- Pipe the results of the same command you used to get the 100 most recent events from the **System** event log of type warning or error to a file using the appropriate “**Out**” command.

► **Task 4: Send command output to a grid and sort and filter the results**

- Pipe the results of the same command you just used to get event log data to the “**Out**” command that displays a grid view.
- Sort the results in the grid by **InstanceId** in descending order.
- Filter the results in the grid to show only one **InstanceId**.

**Results:** After this exercise, you should know how to use PowerShell commands to redirect output to files or a grid view, and how to sort and filter data in the grid view.

## Lab Review

1. How can you use PowerShell to see what a command will do before you actually do it?
2. What are the two most common categories of members on objects and what are they used for?
3. What command is used to show PowerShell data in a grid?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

### Review Questions

- How many optional parameters does the Ps command have?
- What parameter, if any, of Get-EventLog will make the cmdlet connect to a remote computer?
- What cmdlet or cmdlets exist to stop or start services?
- How many properties does a Service object (which is produced by using Get-Service) have?
- What module might add cmdlets for managing Active Directory?



### Review Questions

1. How many optional parameters does the Ps command have?
2. What parameter, if any, of Get-EventLog will make the cmdlet connect to a remote computer?
3. What cmdlet or cmdlets exist to stop or start services?
4. How many properties does a Service object (which is produced by using Get-Service) have?
5. What module might add cmdlets for managing Active Directory?

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 2

## Understanding and Using the Formatting System

### Contents:

Lesson 1: Understanding the Formatting System	2-3
Lesson 2: Using the Formatting System	2-17
Lab: Using the Formatting Subsystem	2-29

## Module Overview

- List and explain the rules that the shell follows when converting elements into a text display
- Explain what cmdlets can successfully receive the output of a Format- cmdlet
- Identify the configuration files used to specify the shell's default formatting
- Identify the general procedure used to provide custom formatting instructions in XML files
- Format cmdlet output objects using the various Format- cmdlets
- Format cmdlet output objects in a table, including calculated columns
- Convert objects to an HTML representation



You have already learned to use the Windows PowerShell® formatting system at a very basic level. However, the formatting subsystem offers a great deal of additional power and flexibility that you can use to create exactly the output you need. By learning how the formatting system works, you can take advantage of its power and flexibility.

### Module Objectives

After completing this module, you will be able to:

- List and explain the rules that the shell follows when converting objects into a text display.
- Explain what cmdlets can successfully receive the output of a Format- cmdlet.
- Identify the configuration files used to specify the shell's default formatting.
- Identify the general procedure used to provide custom formatting instructions in XML files.
- Format cmdlet output objects using the various Format- cmdlets.
- Format cmdlet output objects in a table, including calculated columns.
- Convert objects to an HTML representation.

## Lesson 1

# Understanding the Formatting System

- List and explain the rules that the shell follows when converting elements into a text display
- Explain what cmdlets can successfully receive the output of a Format- cmdlet
- Identify the configuration files used to specify the shell's default formatting



When you first started using Windows PowerShell, you probably just allowed the shell to format output in whatever way it wanted. The shell does have a rich set of default formatting settings, enabling it to format output in a sensible way without requiring very much additional input from you. Those defaults are, however, completely controllable and customizable. Understanding why and how they work is the key to getting the output you need, with the least possible amount of work on your part.

### Lesson Objectives

After completing this lesson, you will be able to:

- List and explain the rules that the shell follows when converting objects into a text display.
- Explain what cmdlets can successfully receive the output of a Format- cmdlet.
- Identify the configuration files used to specify the shell's default formatting.

MCT USE ONLY. STUDENT USE PROHIBITED

## Formatting Defaults

- Nearly every PowerShell cmdlet produces some sort of element as its output
  - Elements contain data, often more data than is presented
  - Elements and their data can be manipulated
- PowerShell's use of elements enables you to customize the formatting of data, without resorting to low-level text manipulation
  - Elements are configured with default output that exposes limited data
  - Elements can be instructed instead to provide more or less data or to reformat that data into different output
- Directly at the command line you can override defaults, supplement defaults, or redirect output to new formats
  - To do this, you'll need to follow a few rules

### Key Points

Keep in mind that nearly all Windows PowerShell cmdlets work with operating system and other elements that natively contain a great deal of information. It wouldn't be practical to try and display all of that information—most computer screens simply don't provide enough room, and having to scroll horizontally to find the information you need would be cumbersome. Therefore, the shell is configured to display only a subset of information by default, and it displays that information using a layout that's best-suited for the information being shown.

For example, a process natively contains dozens of attributes, each of which imparts some potentially-valuable piece of information. There is really no efficient way to display all of that information in an easy-to-use format, so the default output of the Get-Process cmdlet doesn't try. Instead, the shell is configured to display only seven attributes of the individual process and to do so in a relatively straightforward and easy table format.

These defaults are provided with the shell, but they are not hardcoded. In other words, with a little bit of extra work, you can override these defaults and even replace or supplement them with new defaults. You receive the defaults in various XML-formatted configuration files that are installed along with Windows PowerShell.

All of the shell's formatting defaults are based on the *type name* of whatever element is being displayed. You can see the type name by piping those elements to the Get-Member cmdlet; when the pipeline delivers a number of different types of elements, the shell typically selects a default output format based on the type name of the first element in the pipeline.

When the shell needs to use a formatting default, it follows a number of rules to obtain that default.

## Rule 1: Is There a Predefined View?

- For each displayed element, PowerShell will first check to see if a predefined format view has been registered for the type name
  - Views are defined in special XML configuration files
  - These files have a .format.ps1xml extension
  - To see these files, use this command:

**Dir \$pshome**
- The shell loads these files each time it starts, storing view information for the duration of the shell session
- Do not modify predefined views. They are digitally signed and will not load if they have been modified in any way
- Views contain formatting information, such as layout customizations, column labels and widths, script code, etc.

### Key Points

After the shell has a type name of the element it needs to display, it first checks to see whether a format view has already been registered for that type name.

Views are defined in special XML configuration files; several of these files are installed along with Windows PowerShell. Try running this command:

```
Dir $pshome
```

The files with a .format.ps1xml filename extension all contain predefined formatting views. The shell loads these files each time it starts and stores the view information in memory for the duration of the shell session.

**Note:** The .format.ps1xml files included in the shell's installation folder are provided by Microsoft®, and contain a Microsoft digital signature. These specific files will load into the shell even if scripting is disabled (something you will learn about in Module 6), so that the shell can have a reasonable set of defaults. The digital signature ensures that the files cannot be easily changed without you knowing; if the files are changed in *any way*, even accidentally, the signature will be invalid and the shell will stop loading the changed file or files automatically.

If the shell needs to display an element for which a view has already been registered, the shell will use that view. If multiple views have been registered for a type name, the shell will use the first view registered in memory, which means there is some significance to the order in which the view XML files are loaded into memory because the *first* file loaded will take precedence over subsequent files.

Views can be very simple, defining a list, table, wide, or custom layout. They can also be quite complex, defining column labels and widths. They can even contain script code that calculates or retrieves the values for certain table columns.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Finding Format Files

- See the predefined view for a specific element type name



### Key Points

In this demonstration, you'll see the predefined view for a specific element type name.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\Finding Format Files.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Rule 2: Is There a Property Set?

- Sometimes predefined views do not exist
  - When they don't, the shell looks to see if a DefaultDisplayPropertySet type extension has been registered for the type name
  - This type extension is also stored in an XML file, this time with a .types.ps1xml extension
  - To see these files, again use this command:

**Dir \$pshome**
- Do not modify type extensions either. They are digitally signed, and will not load if they have been modified in any way
- Many type extensions ship by default with PowerShell. You can also create your own to customize your display environment

### Key Points

If a predefined view *does not exist*, the shell continues to a second rule. This rule determines which attributes of the elements will be displayed.

A predefined view already specifies which attributes will be displayed, so if the first rule is met, this second rule is unnecessary. However, without a predefined view, the shell needs to decide which attributes to display.

To make this decision, the shell looks to see whether a *DefaultDisplayPropertySet* type extension has been registered for the type name being displayed.

Type extensions are stored in XML files, much like formatting views. If you run this command:

```
Dir $pshome
```

you see the XML files that define the type extensions. These files have a .types.ps1xml filename extension rather than a .format.ps1xml filename extension.

**Note:** Like the .format.ps1xml files, the .ps1xml files in the shell's installation folder are digitally signed by Microsoft and should never be modified.

The shell ships with type extensions for many different element type names. However, there are many kinds of type extensions; the DefaultDisplayPropertySet is only one of many type extensions available. Just because a type name appears in a .ps1xml file doesn't necessarily mean that a DefaultDisplayPropertySet extension has been registered.

However, if one has been registered, the shell uses the attributes named in the extension to make the next decision. If no DefaultDisplayPropertySet has been registered for the type name being displayed, the shell uses *all* of the element's attributes to make the next decision.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Finding a DefaultDisplayPropertySet

- See how the shell locates a DefaultDisplayPropertySet type extension



### Key Points

In this demonstration, you will see how the shell locates a DefaultDisplayPropertySet type extension.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\Finding a DefaultDisplayPropertySet.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

### Rule 3: Table or List?

- Finally, the shell determines how many attributes it needs to display
  - This can be derived from DefaultDisplayPropertySet
  - This can be derived from a predefined formatting view
  - Or, this can simply be all the element's attributes
- Default behavior:
  - If the shell needs to display four or fewer attributes, it defaults to using a table
  - If the shell needs to display five or more attributes, it default s to using a list
- Once this determination is made, the shell creates the final display, internally calling Format-List or Format-Table

#### Key Points

Next, the shell simply asks itself how many attributes it is displaying—obtained either from a DefaultDisplayPropertySet or by using all of the elements' attributes.

**Note:** If a predefined formatting view was used from Rule 1, Rule 3 is never considered. The view indicates what kind of layout—table, list, wide, or custom—will be used.

If the shell needs to display four or fewer attributes, it does so in a table. If it's displaying five or more attributes, it does so in a list. This rule helps to ensure that a table fits into a typical console window using the default console window size.

Once the shell decides which layout—table or list—to use, it begins creating the final display. It does so by internally calling either the Format-List or Format-Table cmdlet, piping in the elements that need to be formatted.

## Out-Default

- The Out-Default cmdlet
  - Is always a component of pipeline output, even if it isn't specifically called
  - Sends whatever elements are in the pipeline to the default display device. This device is typically Out-Host
  - Is the reason why cmdlet output appears in the console window by default
- As a result, these cmdlets are functionally the same:

```
Get-Process  
Get-Process | Out-Default  
Get-Process | Out-Host
```

- You generally do not need to invoke Out-Default. However you need to use other cmdlets (like Out-File) to redirect output

### Key Points

Let's briefly discuss how the formatting system is accessed by the shell.

At the end of each command pipeline is a cmdlet called **Out-Default**. It's always there, even if you don't specifically type it. Its job is to take whatever is in the pipeline at that point and send it to the default display device, which is normally Out-Host.

If you run a command like this:

```
Get-Process
```

you are actually running, without realizing it, the following:

```
Get-Process | Out-Default
```

And in the normal Windows PowerShell ISE or console window, that's functionally the same as running this:

```
Get-Process | Out-Host
```

That's why cmdlet output appears in the console window by default: the output is being internally piped to Out-Default, which is forwarding that output to Out-Host.

But what about when you manually specify another destination, perhaps by using a cmdlet such as Out-File?

```
Get-Process | Out-File c:\procs.txt
```

Most Out- cmdlets don't produce any output, meaning they don't place anything into the pipeline. (Out-String is a special case and is an exception to this general rule.) So even though the preceding command is outputting to a file, the real pipeline—even if you don't realize it—looks like this:

```
Get-Process | Out-File c:\procs.txt | Out-Default
```

And what is functionally happening is this:

```
Get-Process | Out-File c:\procs.txt | Out-Host
```

You don't see anything on the screen when you run that command because Out-File isn't producing any output. It's accepting input and writing a text representation of that input to a file, but after Out-File completes there is nothing in the pipeline. Out-Default is still executing, but it's being given no input to work with, and so nothing appears on the screen.

MCT USE ONLY. STUDENT USE PROHIBITED

## Out- and Format-

- The Out- cmdlets

- Are not capable of understanding normal elements, such as processes and services
- Are designed to use special formatting elements, which are produced by a Format- cmdlet
- As a result, these cmdlets are functionally the same:

**Get-Process**

**Get-Process | Out-Default | Out-Host | Format-Table | Out-Host**

- The formatting system internally determines which Format- cmdlet is called. This determination is based on the registered view for the element.
- Caution: Once a Format- cmdlet executes, your original output is lost. This happens because the cmdlet produces a formatting element.

### Key Points

The Out- cmdlets are technically not capable of understanding normal things like processes and services. The Out- cmdlets are designed only to deal with special formatting instructions, and those instructions can be produced only by a Format- cmdlet.

When an Out- cmdlet finds itself dealing with something other than formatting instructions, it automatically engages the shell's formatting system, including the three rules covered previously. The shell executes a formatting cmdlet internally, and that cmdlet passes its output back to the original Out- cmdlet. So when you run this:

Get-Process

the shell is actually doing something more like this under the hood:

Get-Process | Out-Default | Out-Host | Format-Table | Out-Host

The formatting system determines which Format- cmdlet gets called. In this example, because the output of Get-Process has a registered formatting view that specifies a table layout, the Format-Table cmdlet is used.

You can learn a couple of important lessons from the shell's behavior:

- Once a Format- cmdlet executes, your original output is lost because the Format- cmdlet produces formatting objects. Compare the output of this:

Get-Process | Get-Member

to this:

Get-Process | Format-Table | Get-Member

- And notice that the Format-Table cmdlet has consumed the process elements and produces its own formatting instructions, which makes sense only to an Out- cmdlet. Therefore, a Format- cmdlet, if used, must usually be the last cmdlet on the command line.
- Out- cmdlets, as noted previously, don't produce output. If an Out- cmdlet is used, it's almost always the last cmdlet on the command line (Out-String being the only exception). If an Out- cmdlet and a Format- cmdlet are both used, the Format- cmdlet is next-to-last on the command line, and the Out- cmdlet is last.

All Out- cmdlets are capable of consuming the same formatting instructions. So these two commands:

```
Get-Process | Fl * | Out-Host  
Get-Process | Fl * | Out-File c:\proclist.txt
```

produce identical-looking output, but for differences in the width of the output (meaning some information may wrap or be truncated) and the fact that one outputs to the screen, and the other outputs to a text file.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Output of a Format- Cmdlet

- See the differences between typical cmdlet output and the output of a Format- cmdlet



### Key Points

In this demonstration, you will see the differences between typical cmdlet output and the output of a Format- cmdlet.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\Output of a Format Cmdlet.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Multiple Views

- PowerShell uses, by default, the first predefined view it finds for a type name
- However, you can manually specify an alternate predefined view by using the `-view` parameter:

**Get-Process | Format-Table -view priority**

- Important:** You can reference only views that actually exist. For the preceding command to work, a view called "priority" must be defined for Format-Table
  - There is no way to list the available views for a type name
  - This is why you need to manually search .format.ps1xml files to locate alternate views and learn their names

### Key Points

You can create a formatting view file (a .format.ps1xml file) that contains more than one view for a given type name. In fact, the DotNetTypes.format.ps1xml file actually contains more than one registered view for the System.Diagnostics.Process type name.

Whenever the formatting system is called upon, it uses by default the *first* view registered for a type name. So this command:

Get-Process

produces output identical to that of this command:

Get-Process | Format-Table

The output is identical because in the first case, you have not specified a view or layout. The shell follows Rule 1, locates a predefined view in DotNetTypes.format.ps1xml, and uses the first predefined view it finds for the System.Diagnostics.Process type name. That view happens to be a table. In the second example, you specified a table layout, and so the shell used the first predefined table view that it found, which happened to be the same view from the first example.

However, you can manually specify predefined views other than the first one, by using the `-view` parameter of a Format- cmdlet. There is one rule: The view you specify must correspond to the layout of the Format- cmdlet you use. In other words, if a predefined view named *Diagnostics* exists and is a table-style view, you must use the Format-Table cmdlet. If you tried to run this:

Get-Process | Format-List -view diagnostics

and a list-style view named *diagnostics* didn't exist, you would see an error, even if a table-style view of that same name did exist.

MCT USE ONLY. STUDENT USE PROHIBITED

For example, this command works:

```
Get-Process | Ft -view priority
```

because a predefined table-style view has been registered using the name Priority. However, this command doesn't work:

```
Get-Process | Fl -view priority
```

because the Format-List cmdlet is unable to use the table-style view named Priority.

**Note:** There is unfortunately no way to list the views available for a given type name. You have to search through the .format.ps1xml manually to locate alternate views and learn their names.

## Lesson 2

# Using the Formatting System

- Identify the general procedure used to provide custom formatting instructions in XML files
- Format cmdlet output elements using the various Format- cmdlets
- Format cmdlet output elements in a table, including calculated columns
- Convert elements to an HTML representation



Now that you know how the formatting system works by default, you can start to manipulate it directly for more custom results.

### Lesson Objectives

After completing this lesson, you will be able to:

- Identify the general procedure used to provide custom formatting instructions in XML files.
- Format cmdlet output objects using the various Format- cmdlets.
- Format cmdlet output objects in a table, including calculated columns.
- Convert objects to an HTML representation.

MCT USE ONLY. STUDENT USE PROHIBITED

## Customizing Defaults

- You can create custom formatting views by creating your own .format.ps1xml files. (*Note: This process is beyond the scope of this class.*)
  - New files will register new or supplemental defaults, or new named views in the shell
- It is easiest to use Microsoft's existing .format.ps1xml files as examples for creating your own files
  - Remember: Do not modify the Microsoft-signed .format.ps1xml files, or they will not load
- Once a new file is created, you will use the Update-FormatData cmdlet to load your file into the shell
  - This cmdlet registers the views in your file for the currently-running shell session. It will need to be run in each session.
  - Alternatively, you can run this command within a PowerShell profile script to automatically load it into each session. You will learn more about profile scripts in a later module.

### Key Points

You can create your own .format.ps1xml files to provide new or supplemental defaults, or new named views, in the shell.

**Note:** Do not modify the Microsoft-signed .format.ps1xml files provided with Windows PowerShell.

You do this by creating the necessary XML-formatted file. If you like, you can use the Microsoft-supplied files as examples, being careful not to modify them. Once you have created the necessary file, use the Update-FormatData cmdlet to load your files into the shell, and to register the views in your files with the formatting system.

When you run this cmdlet, you must specify a path to your XML file. To specify that path:

- Use the –append parameter to have your file load after the existing files in the shell's memory. This is good for loading additional views that will be accessed by name, but if the shell already contains a registered view for the same type name, the shell will continue to use its existing view as a default.
- Use the –prepend parameter to have your file load ahead of the existing files in the shell's memory. This means that the views in your file will be used as the shell's new defaults for those object type names.

**Note:** A complete explanation of how to build .format.ps1xml files is beyond the scope of this course. There are books on the market that do cover this topic in more depth, including *Windows PowerShell v2.0: TFM* by Don Jones and Jeffery Hicks.

Note that any custom formatting information remains in-memory only for the duration of the shell session; the shell doesn't automatically reload your custom files. If you have a file that you want available

in every shell session in the future, you can create a Windows PowerShell profile script that runs Update-FormatData each time you open a new shell window. You will learn more about profile scripts in a later module.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Customizing Defaults

- See how to create and load a custom formatting file into the shell



### Key Points

In this demonstration, you will see how to create and load a custom formatting file into the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\Customizing Defaults.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Using the Format Cmdlets

- Creating custom .format.ps1xml files can be difficult
- It can be easier to simply use existing parameters of the Format- cmdlets to customize your needed output
  - Format-Table
  - Format-List
  - Format-Wide
  - Format-Custom
- The help files for these cmdlets document their usage

**Take time now to read through the full help files for these four cmdlets.**

### Key Points

Aside from creating custom .format.ps1xml files, you can also use parameters of the Format- cmdlets to customize output on an ad hoc basis.

- Format-Table offers the most options for customizing table-style output.
- Format-List also offers rich options for creating custom output.
- Format-Wide provides fewer customization options because it's designed to display only a single attribute.
- Format-Custom is a complex cmdlet that generally relies on a predefined view. However, it does have some uses when you want to see a breakdown of an element's attributes.

The best way to learn how to use the Format- cmdlets—as with any other shell cmdlet—is to read their help files.



Take some time now to read through the help files on Format-Table, Format-List, and Format-Wide. Read the full help, including usage examples, to get a better idea of what these cmdlets can do.

## Custom Columns

- You can use `Select-Object` to add custom attributes to an object.

- For example , to add the `ComputerName` attribute to a computer element that already had a `Name` attribute , use:

```
Get-ADComputer -filter * |  
Select  
*,@{Label='ComputerName';Expression={$_.Name}}
```

- This command gathers all of the original attributes through the "\*" wildcard, then adds a new one using "Label"
  - "Expression" defines the value that the attribute will contain and curly braces are used to designate a script block
  - The special "\$\_" placeholder stands in for whatever `Select-Object` was given —in this case, a set of computer names
- There are additional and very powerful uses of this syntax in your student guide. **Review those uses now because you will use them regularly.**

### Key Points

You can use the `Select-Object` to add custom properties to an object. For example, to add a `ComputerName` attribute to a computer element that already had a `Name` attribute, you might run a command like this:

```
Get-ADComputer -filter * |  
Select *,@{Label='ComputerName';Expression={$_.Name}}
```

That `Select-Object` command is accessing all of the original attributes of the computers that were retrieved by `Get-ADComputer`—the \* wildcard makes that happen. In addition, `Select-Object` is adding a new attribute to those computers. The new attribute has a `ComputerName` label, defined by the *Label* portion of the command's attribute-addition construct. The *Expression* portion defines the value that the attribute contains. The Expression is assigned a *script block*, delimited by curly braces. Within that script block, the special `$_` placeholder stands in for whatever `Select-Object` was given—in this case, computers. Following `$_` with a period tells the shell that you want to refer to an existing attribute—in this case, the `Name` attribute of the computers.

The `Format-Table` cmdlet accepts the same syntax for creating custom columns. That means, if all you want to do is add a custom column to a table, rather than adding a custom attribute to the object, you can do this:

```
Get-ADComputer -filter * |  
Ft DnsHostName,Enabled,@{Label='ComputerName';Expression={$_.Name}}
```

Notice that the same kind of construct is used to create a custom column, and the `DnsHostName` and `Enabled` attributes of the computers are included in their own columns.

MCT USE ONLY. STUDENT USE PROHIBITED

**What is the difference?** If you need to modify an element in the pipeline and then pipe it on to another cmdlet, you use Select-Object to add a custom attribute. You will learn more about using that technique in a later module. If you simply want to create a custom column in a table, you use Format-Table. That said, there is no harm in modifying something by using Select-Object and then piping that something to Format-Table to display a table. Both techniques provide similar results. Simply keep in mind that Format-Table usually needs to be the last cmdlet on the pipeline, so if you choose to use it, you can't pipe its output to anything but an Out- cmdlet.

The Expression portion can also include mathematical expressions, using the + - \* or / operators (which correspond to addition, subtraction, multiplication, and division). For example, you might choose to add the values of two attributes:

```
Get-WmiObject Win32_LogicalDisk |  
    Ft DeviceID,Size,  
        @{Label='SpaceUsed';Expression={$_.Size - $_.FreeSpace}}
```

Here, Format-Table is asked to create columns for the DeviceID and Size attributes of the logical disks that were retrieved by Get-WmiObject. In addition to those two columns, it's also creating a third column named *SpaceUsed*. That column contains a value equal to the disks' size, less the amount of free space. You can see that the subtraction operator is used to perform the mathematical calculation.

**Note:** The technique of using `$._` as a placeholder to refer to things, and following it with a period to access attributes, is a core Windows PowerShell technique. You'll see this syntax over and over and should get comfortable with it.

The Expression portion of a custom column can be almost any Windows PowerShell script or command. In this example, it was simply accessing the Name attribute of whatever was piped into Format-Table, but you could specify something much more complex. For example:

```
get-adcomputer -filter * |  
    ft dnshostname,@{  
        Label="OSVersion";  
        Expression={  
            gwmi win32_operatingsystem -comp $._.Name).caption  
        }  
    }
```

This is a complicated command. Here's what's happening:

- The Get-ADComputer cmdlet is retrieving all of the computers in the domain (use caution, because in a large domain, this may be resource-intensive).
- The computers are being piped to Format-Table.
- Format-Table is displaying the DNSHostName attribute.
- Format-Table is also defining a custom column named OSVersion. The contents of this column is an expression that executes the Get-WmiObject cmdlet (which uses the alias gwmi).
- Get-WmiObject is connecting to the computer name specified in the Name attribute of the input to Format-Table—that is, the name of the computer that we're displaying.
- The Get-WmiObject cmdlet is inside parentheses, meaning that the cmdlet executes, and the parentheses represent whatever the result of the command is. In this example, the result is information about the computer's operating system.

- Outside the parentheses, a period indicates that we want to access an attribute of the resulting WMI information. In this example, we're accessing the Caption attribute of the operating system, which is a text description of the operating system's name and version.

The output for a single computer looks something like this:

```
dnshostname          OSVersion
-----
server-r2.company.pri Microsoft Windows Server 2008 R2 Standard
```

**Note:** This is an advanced example, and it uses a couple of techniques that you will explore more fully in upcoming modules. WMI, for example, will be covered later.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Customizing Formatting

- Learn how to use Format- cmdlets to customize the output of cmdlets



### Key Points

In this demonstration, you will learn how to use Format- cmdlets to customize the output of cmdlets.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\Customizing formatting.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## HTML Output

- Sometimes, you want to format data so it can be viewed in a Web browser
- The ConvertTo-HTML cmdlet...
  - Converts elements into an HTML-based table
  - Itself does not write HTML to a file, but instead places string elements into the pipeline
  - You need to pipe its results to Out-File to create an HTML file that can be viewed in the browser

```
Get-EventLog Security -newest 20 |  
ConvertTo-HTML | Out-File events.htm
```

- Parameters of ConvertTo-HTML enable customization of the resulting HTML page
- Note: Do not pipe a Format- cmdlet to ConvertTo-HTML

### Key Points

Sometimes, you want to output cmdlet information in a way that can be viewed in a Web browser. The ConvertTo-HTML cmdlet converts pipeline elements into an HTML-based table. The cmdlet doesn't write that HTML to a file; it simply places HTML text into the pipeline. You can, however, pipe that into a file by using Out-File. A basic use of this cmdlet is as follows:

```
Get-EventLog Security -newest 20 | ConvertTo-HTML |  
Out-File events.htm
```

Don't pipe the output of a Format- cmdlet to ConvertTo-HTML; remember that Format- cmdlets output formatting instructions, and *those* are what will be converted to HTML. If you want to see the confusing-looking results, try this command:

```
Get-Process | Format-Table | ConvertTo-HTML | Out-File confusing.htm
```

The HTML produced by ConvertTo-HTML is well-formed, clean HTML, which means it does not contain formatting information. The result is a plain-looking file. However, by using parameters of ConvertTo-HTML, you can specify additional information, including:

- Text to be placed before the HTML table
- A title for the page
- Text to be placed after the HTML table
- A Cascading Style Sheet (CSS) link, which can be used to specify custom formatting options such as font, color, and so forth

**Note:** As always, read the help for the cmdlet to learn more about its options and capabilities.

## Terminology Reminder

- Remember that PowerShell uses formal terms to describe the elements in the pipeline, as well as their attributes
  - The word *element* was used earlier to refer to things that a cmdlet places in the pipeline
  - The more formal word for elements is *objects*
  - Objects have *attributes*, which are characteristics of that objects
  - The more formal word for attributes is *properties*
- These informal words were introduced earlier to ease your introduction into PowerShell
  - However, you may want to use the formal words as you grow in experience because it makes communication easier

### Key Points

Remember that Windows PowerShell uses formal terms to describe the elements in a pipeline and to describe the attributes of those elements. You learned about these first in the previous module. These are just words; they don't have any inherent value. But you should start to use the formal terms regularly because you'll find it easier to carry on discussions with other shell users.

In this module, the word *element* was used to refer to the things that a cmdlet places into the pipeline. For example, the Get-Process cmdlet places process elements into the pipeline. The more formal word for elements is *objects*; so you say that the Get-Process cmdlet places process *objects* into the pipeline.

These objects have attributes. For a process, that might include details such as its name, ID, memory utilization, and so forth. The formal term for these attributes is *properties*; so you say that the process object has a name property, an ID property, and so on.

## Demonstration: ConvertTo-HTML

- Learn how to convert information to HTML and display the results in a Web browser



### Key Points

In this demonstration, you will learn how to convert information to HTML and display the results in a Web browser.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod02\Democode\ConvertTo-HTML.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lab: Using the Formatting Subsystem

- Exercise 1: Displaying Calculated Properties
- Exercise 2: Displaying a Limited Number of Columns
- Exercise 3: Displaying All Properties and Values of Objects
- Exercise 4: Viewing Objects via HTML
- Exercise 5: Displaying a Limited Number of Properties
- Exercise 6: Displaying Objects Using Different Formatting
- Exercise 7: Displaying a Sorted List of Objects

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 20 minutes**

### Estimated time: 20 minutes

You are a system administrator for a company and work with a team of developers. Recently, some users have complained about an in-house developed application. The developers have asked you to monitor the processes associated with the troubled application and let them have easy access to the data. You have decided to provide them with that information via an HTML-based page so they can easily view the information in a Web browser. You also want to customize your view of this information as you view it in the PowerShell console.

### Lab Setup

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

1. Start the **LON-DC1** virtual machine. You don't need to log on but wait until the boot process is complete.
2. Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Displaying Calculated Properties

### Scenario

You need to check the status of processes on your server, and you want to display their status and also show custom properties in a table format. The main tasks for this exercise are as follows:

1. Retrieve a list of processes.
2. Display a table of objects.
3. Select the properties to display.
4. Use calculated properties to show a custom property.

► **Task 1: Retrieve a list of processes**

- Open Windows PowerShell and retrieve a list of running processes.

► **Task 2: Display a table of objects**

- Show the list of running processes formatted as a table.

► **Task 3: Select the properties to display**

- Show the list of running processes formatted as a table showing only the **CPU**, **Id**, and **ProcessName** values.

► **Task 4: Use calculated properties to show a custom property**

- Show the list of running processes formatted as a table showing the **CPU**, **Id**, **ProcessName** values, and a custom property named **TotalMemory**, which is the sum of the physical and virtual memory used for each process.

**Hint:** Just add the values together—don't be concerned if one is displayed in KB and the other in MB.

**Results:** After this exercise, you will have displayed the CPU, ID and ProcessName values of all the running processes. You also will have created a custom property adding the physical and virtual memory using calculated properties.

## Exercise 2: Displaying a Limited Number of Columns

### Scenario

You need to display a list of services on your systems. You need to filter out some of the columns that you don't need when reviewing the status of the services. The main tasks for this exercise are as follows:

1. Retrieve a list of services.
2. Display a specific number of columns from a collection of objects.

► **Task 1: Retrieve a list of services**

- Retrieve a list of services.

► **Task 2: Display a specific number of columns from a collection of objects**

- Retrieve a list of services and display only the **Status** and **Name** properties.

**Results:** After this exercise, you will have displayed a specific number of columns from all the installed services instead of displaying all their default properties.

## Exercise 3: Displaying All Properties and Values of Objects

### Scenario

You often retrieve a list of processes from your systems and need to display all the properties and values of the services and not just the default values normally display. The main tasks for this exercise are as follows:

1. Retrieve a list of processes.
2. Display every property and value for all the objects.

► **Task 1: Retrieve a list of processes**

- Retrieve a list of processes.

► **Task 2: Display every property and value for all the objects**

- Retrieve a list of processes and display all their properties and values.

**Results:** After this exercise, you will have displayed all of the properties of all of the running processes instead of displaying only their default properties.

## Exercise 4: Viewing Objects via HTML

### Scenario

You have been asked by the development team to post a log of entries from one server's application event log. You have decided to automate the gathering of the information and are posting it on an internal Web site for the developers to review. The main tasks for this exercise are as follows:

1. Retrieve a list of event log entries.
2. Convert a list of event log entries to HTML.
3. Use a custom title for an HTML page.
4. View an HTML page in the default Web browser.

► **Task 1: Retrieve a list of event log entries**

- Retrieve the newest 25 events from the **Windows Application** event log.

► **Task 2: Convert a list of event log entries to HTML**

- Retrieve the newest 25 events from the **Windows Application** event log and convert the events as HTML and save the results to a file.

► **Task 3: Use a custom title for an HTML page**

- Retrieve the newest 25 events from the **Windows Application** event log and convert the events as HTML file while changing its title to **Last 25 events** and save the results to a file.

► **Task 4: View an HTML page in the default Web browser**

- View the resulting file from the previous task in the default Web browser (Hint: use **Invoke-Item**).

**Results:** After this exercise, you will have displayed a customized HTML page that contains a listing of the 25 newest events from the Windows Application event log.

## Exercise 5: Displaying a Limited Number of Properties

### Scenario

You need to review the length of certain files in a particular directory and want to use PowerShell to accomplish this task. The main tasks for this exercise are as follows:

1. Display a list of files from a selected folder.
2. Display a table with several different file properties.

► **Task 1: Display a list of files from a selected folder**

- Retrieve a list of all the files that start with *au* and end with *.dll* from the C:\Windows\System32 directory.

► **Task 2: Display a table with several different file properties**

- Retrieve a list of all the files of that start with *au* and end with *.dll* from the C:\Windows\System32 directory and display the **Name**, **Length** and **Extension** properties using a table.

**Results:** After this exercise, you will have displayed a table containing the Name, Length and Extension properties of all the files in C:\Windows\System32 that start with *au* and have the extension *.dll*.

## Exercise 6: Displaying Objects Using Different Formatting

### Scenario

You often use PowerShell to verify the status of processes and services and want to know how to change the way that the information is displayed in the console. The main tasks for this exercise are as follows:

1. Retrieve a list of services.
2. Display a table showing only a few specific properties.
3. Display a table by eliminating any empty spaces between columns.

► **Task 1: Retrieve a list of services**

- Retrieve a list of services.

► **Task 2: Display a table showing only a few specific properties**

- Retrieve a list of services and display only the **DisplayName**, **Status** and **DependentServices** properties using a table.

► **Task 3: Display a table by eliminating any empty spaces between columns**

- Retrieve a list of services and display only the **DisplayName**, **Status** and **DependentServices** properties using a table and eliminating any empty spaces between columns.

**Results:** After this exercise, you will have displayed a table of all the installed services by displaying only the DisplayName, Status, and DependentServices properties. You have also removed any empty spaces between each column.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 7: Displaying a Sorted List of Objects

### Scenario

You use PowerShell to view the status of processes and services and want to modify how some of the columns display data and even possibly create new customized columns. The main tasks for this exercise are as follows:

1. Retrieve a list of services.
2. Display a table while sorting a specific property.

► **Task 1: Retrieve a list of services**

- Retrieve a list of services.

► **Task 2: Display a table while sorting a specific property**

- Retrieve a list of services and display a table showing the services by sorting on the **Status** property.

**Results:** After this exercise, you will have displayed a table of the installed services sorted by their Status property.

## Lab Review

1. It would be nice to be able to format the color of each row in some cases. Does Format-Table appear to provide the ability to change the color of rows?
2. What might happen if I used Select-Object in the pipeline, after I've used Format-Table (...|format-table <something>|select-object <something>)? Would that work?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- What parameter can be used to keep Format-Table from wasting space when only two or three columns are displayed?
- What happens when you try to display more table columns than will fit on the screen? Is there a way to keep column information from being truncated?



### Review Questions

1. What parameter can be used to keep Format-Table from wasting space when only two or three columns are displayed?
2. What happens when you try to display more table columns than will fit on the screen? Is there a way to keep column information from being truncated?

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 3

## Core Windows PowerShell® Cmdlets

### Contents:

<b>Lesson 1:</b> Core Cmdlets for Everyday Use	3-3
<b>Lab A:</b> Using the Core Cmdlets	3-22
<b>Lesson 2:</b> Comparison Operators, Pipeline Filtering, and Object Enumeration	3-29
<b>Lab B:</b> Filtering and Enumerating Objects in the Pipeline	3-42
<b>Lesson 3:</b> Advanced Pipeline Techniques	3-49
<b>Lab C:</b> Using Pipeline Parameter Binding	3-63

## Module Overview

In this module, you will learn to:

- Recognize Windows PowerShell's core cmdlets
- Sort, group, count, measure, and select objects in the pipeline
- Import and export objects from various sources
- Compare sets of objects



Many of the cmdlets built into Windows PowerShell® don't perform useful administrative tasks by themselves. Instead, these cmdlets work in conjunction with other, more task-focused cmdlets to help you accomplish specific tasks, put data into a different order or format, and so forth.

At first, it seems that these cmdlets are less interesting, but they are really some of the most important Windows PowerShell cmdlets you can learn. These cmdlets will prove useful no matter what products or technologies you are managing through Windows PowerShell.

### Module Objectives

After completing this module, you will be able to:

- Recognize Windows PowerShell's core cmdlets.
- Sort, group, count, measure, and select objects in the pipeline.
- Import and export objects from various sources.
- Compare sets of objects.

## Lesson 1

# Core Cmdlets for Everyday Use

This lesson teaches you to:

- Recognize Windows PowerShell core cmdlets
- Sort objects in the pipeline
- Group objects in the pipeline
- Count and measure objects in the pipeline
- Select specific properties of objects in the pipeline
- Select a subset of the objects in the pipeline
- Import and export objects to and from delimited files
- Import and export objects to and from XML files
- Compare sets of objects, including objects imported from a delimited or XML file



One of the most important concepts in Windows PowerShell is that of *objects*, which you first learned about in the previous module. Windows PowerShell does not simply display text output when you run a command; instead, the shell generates objects and places them into a pipeline. The core cmdlets you'll learn in this lesson manipulate those objects within the pipeline, to sort them, group them, measure them, and so on. The ability to connect several of these cmdlets together in the pipeline is part of what enables Windows PowerShell to accomplish complex administrative tasks with relatively straightforward, one-line commands rather than forcing you to write a lengthy script for every task that you complete.

### Lesson Objectives

After completing this lesson, you will be able to:

- Recognize Windows PowerShell core cmdlets.
- Sort objects in the pipeline.
- Group objects in the pipeline.
- Count and measure objects in the pipeline.
- Select specific properties of objects in the pipeline.
- Select a subset of the objects in the pipeline.
- Import and export objects to and from delimited files.
- Import and export objects to and from XML files.
- Compare sets of objects, including objects imported from a delimited or XML file.

MCT USE ONLY. STUDENT USE PROHIBITED

## The Core Cmdlets

- Data Manipulation

- Sort-Object
- Group-Object
- Measure-Object
- Select-Object
- Compare-Object

**These cmdlets manipulate data from other cmdlets**

- Importing and Exporting Data

- Import-CSV
- Export-CSV
- Import-CliXML
- Export-CliXML

**These cmdlets import/export data from other formats**

### Key Points

While many of the shell's built-in cmdlets could be considered *core*, only a few of the cmdlets are designed to work generally with whatever objects might be placed in the pipeline by other cmdlets. The cmdlets that do so typically have the noun *object* as part of their cmdlet name. The cmdlets in this lesson are:

- Sort-Object
- Group-Object
- Measure-Object
- Select-Object
- Compare-Object

Some additional core cmdlets are also provided to import and export data from and to various text file formats. These cmdlets are:

- Import-CSV
- Export-CSV
- Import-CliXML
- Export-CliXML

Keep in mind that, aside from the Import- cmdlets, none of these cmdlets generate objects of their own. Instead, they are meant to accept objects that are generated by other cmdlets, do something with those objects, and then pass those objects (which may have been modified or filtered in some way) to the pipeline and then to whatever cmdlet might be next in the pipeline.

## Objects and Members

- An Object...
    - Is a general software term that typically refers to some self-contained piece of functionality
  - A PowerShell cmdlet...
    - Often consumes an object to accomplish some task
    - Typically produces an object as a result, or manipulates other objects to produce an object
    - For example,
- Object -> Cmdlet -> Object**
- The Event Log -> Get-EventLog -> An Event Log View**
- An object is superior to a simple text result, because it can be sorted, grouped, measured, selected, and so forth

### Key Points

This is a good time to re-examine the fact that Windows PowerShell cmdlets don't produce simple text as their output. Instead, cmdlets produce objects.

*Object* is a general software term that typically refers to some self-contained piece of functionality that exposes information about itself in the form of *properties*, and offers *methods* that tell the software to perform some task or function.

Many Windows PowerShell cmdlets, especially those that use the Get- verb in their cmdlet name, produce objects and place them into the pipeline:

```
Get-EventLog Security -newest 20
```

By piping those objects to another cmdlet, you can manipulate them in the pipeline:

```
Get-EventLog Security -newest 20 | Sort-Object
```

Some of the core cmdlets in this lesson simply manipulate those input objects and output them to the pipeline. Other core cmdlets consume the object you pipe in, and produce a new kind of object that is output to the pipeline. You'll explore the differences as you examine each of the core cmdlets.

## Sorting Objects

- The Sort-Object cmdlet...
  - Enables you to change the order in which objects are listed.
  - Takes any type of object as input.
  - Can specify the name of one or more properties to be sorted.
  - Uses ascending sort order by default.
  - Uses the -descending parameter to sort descending.
  - Operates on actual property values, which are not always displayed. PowerShell can convert values to text strings.

**Get-Service | Sort-Object status  
Get-Service | Sort-Object status -descending**

**Get-Service | Sort-Object status, name**

- Sort is a built-in alias for Sort-Object.

### Key Points

When you run a cmdlet, the cmdlet itself determines the order in which its output objects are listed. Retrieving a list of processes, for example, displays the processes in alphabetical order by process name. A list of event log entries, on the other hand, is normally listed in chronological order.

The cmdlet's default order may not always be what you want. For instance, finding a list of all stopped services is difficult when the services are listed alphabetically by service name.

Sort-Object enables you to change the order in which objects are listed. This cmdlet can receive any type of object as its input, and you specify the name of one or more properties on which you would like the objects sorted. The sort order is ascending by default, and you can specify descending by adding the -descending parameter. Your computer's operating system culture determines the actual sort order, so sort results may be slightly different from one computer to another if they are configured to use a different culture. For example:

**Get-Service | Sort-Object status**

Or, to sort the objects in descending order:

**Get-Service | Sort-Object status -descending**

Sort is a built-in alias for Sort-Object, and can be used in place of the full cmdlet name:

**Get-Service | Sort status**

Note that the property names are not case-sensitive; *status*, *Status*, and *STATUS* are all treated the same. Also, when Sort-Object sorts a text property, such as a process or service name, it performs a case-insensitive sort by default. Run Help Sort-Object to see other options for using this cmdlet.

The Sort-Object cmdlet can seem to run slowly when you pipe in a very large collection of objects. That's because the cmdlet must wait until it receives every object before it can create the final, sorted output. Once the last object is received, it is sorted, and then the cmdlet produces the sorted output.

You can specify multiple properties by providing a comma-separated list of property names. For example, if you want a list of all services sorted by status, but sorted by service name within each status, you might run:

```
Get-Service | Sort status, name
```

Be aware that Sort-Object operates on the actual property values of an object, but Windows PowerShell does not always display those actual property values. The status property of a service is a good example. Internally, the status is represented by a number; however, Windows PowerShell's extensible type system automatically translates that number to a text string for you. So instead of seeing a status of *0* or *1*, you see a status of *Stopped* or *Running*. If you pay close attention, you will notice that the following command sorts *Stopped* before *Running*, even though *R* would normally come *before S*, and not after it:

```
Get-Service | Sort status
```

This behavior is caused by Sort-Object operating on the underlying property value and sorting *0* before *1*, as is appropriate. It is not until the shell displays those objects that *0* turns into *Stopped* and *1* turns into *Running*.

MCT USE ONLY. STUDENT USE PROHIBITED

## Grouping Objects

- The Group-Object cmdlet...
  - Provides a way to place objects into different groups
  - Enables manipulation of results by group
  - Examines a designated object property and creates a new group of objects for each property value it encounters
  - Is useful only when object properties have repetitive values
  - Can be used for counting the number of objects in a group

**Get-Service | Group-Object status**

### Key Points

Sometimes you may want to place objects into different groups, so you can work with different groups in different ways. The Group-Object cmdlet enables you to do this. It examines a designated object property, and creates a new group of objects for each property value it encounters. For example, this command:

```
Get-Service | Group-Object status
```

will normally create two groups of objects: One group with a *Running* status, and the other group with a *Stopped* status. You could potentially see other groups for *Starting*, *Stopping*, and other service statuses if you have services in one of those conditions at the time you run this command.

Group-Object is normally useful only when you have object properties with repetitive values. For example, the following command is less useful:

```
Get-Service | Group-Object name
```

Because each service object has a unique name, Group-Object is forced to produce a new group for each service, with each group containing only one service.

One benefit of Group-Object is that its output shows you how many objects are in each group, allowing you to see quickly how many services are stopped, how many processes are responding to Windows, and so forth.

## Demonstration: Sorting and Grouping Objects

- Using Sort-Object in the shell
- Using Group-Object in the shell



### Key Points

In this demonstration, you will learn how to use Sort-Object and Group-Object within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Sorting and Grouping Objects.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Measuring Objects

- The Measure-Object cmdlet...
  - Counts the number of objects piped to it
  - Measures specific aggregate values for numeric properties
    - -average
    - -maximum
    - -minimum
    - -sum
  - Consumes its input objects, meaning that they are no longer in the pipeline
  - Is typically used as the last PowerShell command in a series of pipelined commands

**Get-Process | Measure-Object**

**Get-Process | Measure-Object –property VM  
–average –sum –minimum –maximum**

### Key Points

The Measure-Object cmdlet can count the number of objects that you pipe to it, and it can measure specific aggregate values for numeric property values. At its simplest, you can pipe objects in to get a count of them:

**Get-Process | Measure-Object**

Note that Measure-Object consumes the input objects, meaning they are no longer in the pipeline. Measure-Object outputs a measurement object, which you see as the result of the above command. You can verify that the output of Measure-Object is a measurement object, and not a process object, by piping its output to Get-Member:

**Get-Process | Measure-Object | Get-Member**

This behavior means that your process objects are effectively lost to you after running Measure-Object. In practice, Measure-Object is commonly the last command on the pipeline, simply because there are few scenarios where you would want to pipe its measurement object to another command.

In addition to counting objects, Measure-Object can create aggregate values for numeric property values, such as the VM (Virtual Memory) property of a process object. To do this, specify one or more aggregate values, and the name of the property you want to measure:

**Get-Process | Measure-Object –property VM –average –sum –minimum  
–maximum**

The resulting measurement object will be fully populated. You can see other Measure-Object options by using the Help command to view the help for Measure-Object.

**Note:** It is a common practice to shorten parameter names to save typing, for example, entering “-min” instead of “-minimum”. This practice works fine, but remember that you are typing a truncated parameter name, not a common abbreviation for it. For example, -aver will stand in place of -average, but -avg will not work, even though avg is a common English abbreviation for average.

MCT USE ONLY. STUDENT USE PROHIBITED

## Selecting Objects

- The Select-Object cmdlet...
  - Is used for two distinct purposes:
    1. Selecting a subset of objects: -first, -last, -skip
      - Get-Process | Select-Object -first 10**
    2. Selecting specified properties of an object
      - Get-Process | Select-Object name, ID, VM, PM**
  - Combinations of its two purposes are also useful
    - Get-Process | Select-Object name, ID -first 10**

### Key Points

The Select-Object cmdlet has two distinct purposes, or two ways that the cmdlet can be used. These two ways can be used independently or together, depending on your needs.

### Selecting a Subset of Objects

You can use Select-Object to select a subset of the objects in the pipeline, by using its:

- -first, -last, and -skip parameters.
- -first selects the first specified number of objects in the pipeline.
- -last selects the last specified number of objects in the pipeline.
- -skip skips the specified number of objects and displays the remaining ones.

For example, to select only the first ten processes:

```
Get-Process | Select-Object -first 10
```

To select the ten processes using the most physical memory:

```
Get-Process | Sort-Object PM | Select-Object -last 10
```

When used in this way, Select-Object outputs the same type of objects that were input to it—it may simply output fewer of them. You can verify this behavior by piping the output of Select-Object to Get-Member:

```
Get-Process | Select-Object -first 10 | Get-Member
```

### Selecting Specified Properties

You can also use `Select-Object` to restrict the properties of its output objects. For example, if you are displaying processing and only want to work with the name, ID, VM, and PM properties, you can use `Select-Object` to remove the other object properties:

```
Get-Process | Select-Object name, ID, VM, PM
```

However, when you do so, you are forcing `Select-Object` to output a new, custom object type. The original process objects are consumed and are not output to the pipeline. You can verify this by piping the output to `Get-Member`:

```
Get-Process | Select-Object name, ID, VM, PM | Get-Member
```

This behavior is true anytime you specify a list of properties with `Select-Object`. You can use this technique in conjunction with `-first`, `-skip`, and `-last`, but because there is a property list, `Select-Object` will output custom objects, not the original input objects:

```
Get-Process | Select-Object name, ID, VM, PM -first 10 | Get-Member
```

`Select` is a pre-defined alias for `Select-Object`, and can be used in place of the cmdlet name:

```
Get-Process | Select name, id -first 10
```

## Creating Custom Properties

You can also use `Select-Object` to attach new, custom properties to an object. These custom properties may be simply a new name for an existing property, such as adding a *computername* property to an object that already has a *machinename* property. Custom properties can also contain complex calculations, such as creating a *PercentageFreeSpace* for an object that already has *TotalSize* and *FreeSpace* properties.

Creating custom properties requires you to create a *hashtable*. You will learn more about hashtables later in this course, but for now you can simply memorize the correct syntax:

```
Get-Process | Select name,vm,pm,@{Label="TotalMemory";  
Expression={$_.vm + $_.pm}}
```

Here is how this works:

- The `Select-Object` cmdlet is being told to select the name VM, and PM properties that the process object already has.
- Although `Select-Object` is case-insensitive, it does preserve case. The *vm* property is displayed in lowercase in the final output, because it was specified in all-lowercase in the command.
- The custom property has a label of *TotalMemory*. The value of the property is calculated based on the process object's existing VM and PM properties. The Expression portion of the custom property defines this value as the sum of the current process object's VM property and its PM property; the `$_` variable is a placeholder that refers to "the current object."

**Note:** You can think of `$_` as a *fill in the blank*, similar to the underscored *fill-in blanks* that you might find on a paper form. The shell automatically fills in the blank with the current pipeline object. `$_` can be used only in certain scenarios where the shell is specifically looking for it. The Expression block of a custom property is one of those scenarios.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Measuring and Selecting Objects

- Using Measure-Object in the shell
- Using Select-Object in the shell



### Key Points

In this demonstration, you will learn how to use Measure-Object and Select-Object within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Measuring and Selecting Objects.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Importing and Exporting Data to CSV and XML

- Sometimes data needs to come from or be used by other sources
  - Imported-from/exported-to non-PowerShell aware sources
  - Imported-from/exported-to databases, spreadsheets, and so forth
- PowerShell natively imports/exports data to and from CSV and XML file formats
  - Import-CSV, Export-CSV, Import-CliXML, Export-CliXML

```
Get-EventLog Security -newest 20 | Export-CSV new-events.csv
```

```
Get-Process | Sort VM -desc | Select -First 10 | Export-CSV top-vm.csv
```

```
Import-CliXML procs.xml | Get-Member
```

- Remember: Exported data is no longer a “live” object

### Key Points

Windows PowerShell comes with the ability to read and write comma-separated values (CSV) files and simple XML files. For XML files, the shell uses a well-formed, custom XML layout called Command Line Interface XML, or CliXML.

### Exporting Objects to a CSV File

When you pipe objects to **Export-CSV**, and specify a filename for the export file, the shell examines each of the objects in the pipeline. It writes a comment header as the first line of the export file, documenting the type name of the objects in the pipeline. Then, it writes a header row, listing each property of the objects in the pipeline. From there, each object in the pipeline is written as a row of the CSV file, with each of the object’s properties filled into the appropriate columns of the CSV. Completed files can be easily opened in Microsoft Office Excel, imported into numerous database applications, used as a mail merge source in Microsoft Office Word, and so forth.

A simple use of the cmdlet might look like this:

```
Get-EventLog Security -newest 20 | Export-CSV new-events.csv
```

You could also manipulate objects in the pipeline before exporting them to a CSV file:

```
Get-Process | Sort VM -desc | Select -First 10 | Export-CSV top-vm.csv
```

The Export-CSV cmdlet includes a number of optional parameters you can use to customize its behavior. For example, you can:

- Specify that the type name comment be omitted.
- Specify that the column header row be omitted.
- Specify a delimiter other than a comma, such as a pipe character.

Use the Help command to read the help file for Export-CSV and to learn more about these and other options.

It is important to note that a CSV file is a *flat file* format. That is, it can contain only one dimension of data. Windows PowerShell is not able to turn a complex object hierarchy into a CSV file. For example, a file system folder object can contain other folders as well as files; Export-CSV can't explore this hierarchy. Try examining the file created by this command to see this behavior:

```
Dir C:\ | Export-CSV directories.csv
```

### Importing Objects from a CSV File

The Import-CSV cmdlet is capable of reading a CSV file (or files using other delimiters) and creating static objects that represent the contents of the CSV file. Essentially, each row in the CSV file becomes an object, which is placed into the pipeline. The objects' properties are constructed from the columns in the CSV. If the CSV file includes a header row (the default behavior assumes that the file contains a header row), the objects' property names are taken from this header row.

You can review the help file for Import-CSV to see other options, including the ability to:

- Specify that the CSV file contains no header row.
- Specify a delimiter character other than a comma.

### Importing and Exporting XML Files

The Import-CliXML and Export-CliXML cmdlets work much like the Import-CSV and Export-CSV cmdlets. However, instead of using a flat-file CSV format, these cmdlets use a more robust format that is capable of representing hierarchical data. Try this command:

```
Dir c:\Windows\System32 -recurse | Export-CliXML directories.xml
```

You can open the file in Windows® Notepad, but it may be easier to view in Internet Explorer, which is capable of formatting XML files for better visual display.

Writing an object to XML in this way is called *serializing*; reading an XML file back into the shell and constructing objects again is called *de-serializing*. De-serialized objects are not live objects. That is, they are a snapshot, created at the time they were originally serialized. De-serialized objects no longer have any functionality or methods; they are simply a collection of properties.

To explore this difference, first run this command:

```
Get-Process | Get-Member
```

Make a note of the type name shown, and notice that the member list includes numerous methods.

Now run this command:

```
Get-Process | Export-CliXML procs.xml
```

Now start a new process by opening an application such as Windows Notepad or Windows Calculator, and run this command:

```
Import-CliXML procs.xml
```

The display looks similar to the display created by Get-Process. However, notice that your new process is not listed, because these objects were de-serialized from the XML file, not queried from the operating system. Now run this command:

```
Import-CliXML procs.xml | Get-Member
```

The output type name indicates that this is a de-serialized object, and the objects no longer have methods – they are limited to properties, because that's all the XML file can contain.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with CSV Files

- Use Import-CSV in the shell
- Use Export-CSV in the shell



### Key Points

In this demonstration, you will learn how to use Import-CSV and Export-CSV within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Working with CSV Files.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Comparing Objects

- The Compare-Object cmdlet...
  - Provides the capability of comparing two sets of objects
  - Is used for comparing values of similar properties or object characteristics
  - Provides garbage results if two dissimilar elements are compared

**Correctly using Compare-Object can be tricky.  
Try exporting a snapshot of running services, then change  
a service, and then compare the two:**

```
Get-Service | Export-CliXML service-baseline.xml  
Start or stop a service  
Compare-Object (Get-Service) (Import-CliXML  
service-baseline.xml)
```

### Key Points

Windows PowerShell includes the ability to compare two sets of objects. Comparing objects can be difficult and complex, because objects themselves can be complex. For example, consider an identical process running on two computers, such as Windows Notepad. Certain aspects of the process objects will be identical across the two computers, such as the name property. Other properties, however, will be different, such as the ID property, or the VM and PM properties. Are these objects identical? It depends on exactly why you need to compare them.

One use for comparing objects is for change management. You may, for example, want to create a baseline that describes how a server was initially configured. Later, you may want to compare the server's current configuration to that baseline, to see what has changed. Windows PowerShell offers a cmdlet, Compare-Object (which has an alias, Diff), that can make comparisons easier.

You might start by creating a baseline file. XML is best-suited for this purpose, and this command creates a file with the current configuration of the computer's services:

```
Get-Service | Export-CliXML service-baseline.xml
```

Try making a change to a service, such as starting a stopped service or stopping an unnecessary service. The BITS service is a safe one to manipulate in a lab environment. Then, compare the new configuration to the baseline:

```
Compare-Object (Get-Service) (Import-CliXML service-baseline.xml)
```

In this command, the parentheses cause the Get-Service and Import-CliXML cmdlets to run first. Their output is passed to the two parameters of Compare-Object, which then compares the two sets of objects.

In this instance, Compare-Object will compare all properties of all objects. However, with other types of objects—such as processes, whose memory and CPU values are constantly changing—it would be useless

to compare all object properties, because they will always be different. In those cases, you might want to have `Compare-Object` consider only a subset of the objects' properties for its comparison.

Read the shell's help file for `Compare-Object` to see how to specify properties, as well as to review other options and capabilities of this cmdlet.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Comparing Objects

- Use Compare-Object in the shell



### Key Points

In this demonstration, you will learn how to use Compare-Object within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Comparing Objects.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab A: Using the Core Cmdlets

- Exercise 1: Sorting and Selecting Objects
- Exercise 2: Retrieving a Number of Objects and Saving to a File
- Exercise 3: Comparing Objects Using XML
- Exercise 4: Saving Objects to a CSV File
- Exercise 5: Measuring a Collection of Objects

### Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You work as a system administrator in a small company. The company you work with has not invested in an enterprise-class monitoring solution. As a result, you have to do a lot of performance monitoring and capacity management/trending. One of your duties is to log in to servers and gather some basic statistics, which you do on an almost daily basis. You would like to keep a record of their current status by saving a file locally on the server as a future reference.

#### Lab Setup

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

1. Start the **LON-DC1**, **LON-SVR1**, and **LON-SVR2** virtual machines, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Disable the Windows Firewall on **LON-DC1**, **LON-SVR1**, and **LON-SVR2**. You can do this from **Start | Control Panel | Windows Firewall**. There, click the link to **Turn Windows Firewall on or off**, and **Turn off the Windows Firewall for all network locations**.

**Note:** This is not security best practice and under no circumstances should this be done in a production environment. Using Windows PowerShell doesn't necessarily require the Windows Firewall to be completely disabled. However, purely for the learning purposes of this class, we will disable the firewall on all machines for all labs in order to focus our attentions on PowerShell as opposed to firewall configurations to get a better understanding of the powershell concepts involved.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 1: Sorting and Selecting Objects

### Scenario

You often check the status of processes on your server and would like to sort the processes so you can easily view their status.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows

1. Retrieve a list of processes.
2. Sort a list of processes.
3. Select a certain number of processes.

► **Task 1: Retrieve a list of processes**

- Open **Windows PowerShell** and retrieve a list of running processes.

► **Task 2: Sort a list of processes**

- Retrieve a list of running processes and sort them.

► **Task 3: Select a certain number of processes**

- Retrieve a list of running processes, sort them, and select only the first five.

**Results:** After this exercise, you will have retrieved a list of first 5 services sorted alphabetically by their service name.

## Exercise 2: Retrieving a Number of Objects and Saving to a File

### Scenario

You are currently having problems with an application, and want to check for the last five events in the Windows Application event log.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows

1. Retrieve a list of event logs.
2. Retrieve a certain number of events.
3. Export a certain number of events to a file.

► **Task 1: Retrieve a list of event logs**

- Retrieve a list of the Windows event logs.

► **Task 2: Retrieve a certain number of events**

- Retrieve a list of the five newest/most recent events in the **Windows Application** event log.

► **Task 3: Export a certain number of events to a file**

- Retrieve a list of the five newest/most recent events in the **Windows Application** event log and export the events to a CSV formatted file.

**Results:** After this exercise, you will have exported the five most recent events from the Windows Application event log to a CSV file.

## Exercise 3: Comparing Objects Using XML

### Scenario

You suspect that the status of a service is changing, and want to monitor that status of the service so you can determine if its status has changed.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Retrieve a list of services.
2. Save a list of services to a XML formatted file.
3. Stop a service.
4. Compare services with different statuses.
5. Restart a service.

► **Task 1: Retrieve a list of services**

- Retrieve a list of services.

► **Task 2: Save a list of services to an XML formatted file**

- Retrieve a list of services and save the listing to an XML formatted file.

► **Task 3: Stop a service**

- Stop and thus change the status of the **Windows Time** service ("w32time").

► **Task 4: Compare services with different statuses**

- Retrieve a new list of services and save the listing to an XML formatted file. (Use a different file name than what was used in Task 2.)
- Using the data from the XML formatted files, compare the old and new list of services by comparing the **Status** property.

**Hint:** When you specify a comparison property for the Compare-Object cmdlet, you can include more than one property. To do so, provide list of properties for the cmdlet's -property parameter, inserting a comma between each property. For example, if you tell it only to compare the Status property, you will not be able to see the name of the service. If you include both the State and Name properties, then you will see the name of the service.

► **Task 5: Restart a service**

- Change the status of the **Windows Time** service back to its original state.

**Results:** After this exercise, you will have determined that the Status property of two service objects has changed after the Windows time service was modified.

## Exercise 4: Saving Objects to a CSV File

### Scenario

You want to get a list of processes in a format that could be readable by Microsoft Excel to produce management reports and metrics.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows

1. Retrieve a list of processes.
2. Export a list of processes to a file.
3. Use a non-default separator.

► **Task 1: Retrieve a list of processes**

- Retrieve a list of running processes.

► **Task 2: Export a list of processes to a file**

- Retrieve a list of running processes and export them to a CSV formatted file.

► **Task 3: Use a non-default separator**

- Retrieve a list of running processes and export them to a CSV formatted file, but use a semi-colon ";" as the delimiter.

**Results:** After this exercise, you will have exported a list of processes to a CSV file using the default separator ",", and the non-default separator ";".

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 5: Measuring a Collection of Objects

### Scenario

You are getting reports that there seem to be problems with one of the servers you are monitoring, and want to get a quick report of the CPU time used by processes on the server.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows

1. Retrieve a list of processes.
2. Measure the average, maximum, minimum, and average of a collection.

► **Task 1: Retrieve a list of processes**

- Retrieve a list of the running processes.

► **Task 2: Measure the average, maximum and minimum of a collection**

- Retrieve a list of the running processes and measure the average, maximum, and minimum of the CPU property of the entire collection.

**Results:** After this exercise, you will have measured the average, minimum and maximum values of the CPU property of all the running processes.

## Lab Review

1. How might sorting work with numbers versus strings?
2. Can you think of other delimiters you may want to use with Export-Csv?
3. Is there a difference between saving objects to a variable versus saving them to XML formatted file?
4. Thinking about objects having properties and methods, if an object is saved to a file, is anything lost?

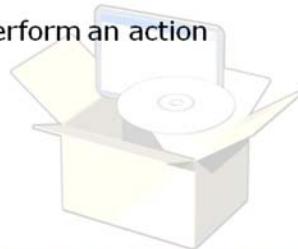
MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 2

# Comparison Operators, Pipeline Filtering, and Object Enumeration

This lesson teaches you to:

- Explain the purpose and use of basic comparison operators
- Explain the purpose and use of Boolean operators
- Interactively compare two values or objects to each other
- Filter objects out of the pipeline based on criteria
- Identify the best place to apply filtering given performance criteria
- Enumerate objects in the pipeline and perform an action with or against each object



One of the other core cmdlets of Windows PowerShell is designed to reduce the number of objects you are working with by removing unwanted objects from the pipeline. The way you typically tell the shell what objects are unwanted is by specifying a comparison expression, which uses one or more comparison operators.

Another core cmdlet is designed to take a collection of objects and enable you to work with each one individually, rather than requiring you to work with them in bulk as is more commonly done in the pipeline.

In this lesson, you will learn to use both of these core cmdlets, and the basic shell comparison operators, to perform several different administrative tasks.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and use of basic comparison operators.
- Explain the purpose and use of Boolean operators.
- Interactively compare two values or objects to each other.
- Filter objects out of the pipeline based on criteria.
- Identify the best place to apply filtering given performance criteria.
- Enumerate objects in the pipeline and perform an action with or against each object.

MCT USE ONLY. STUDENT USE PROHIBITED

## Understanding Comparison Operators

- Comparisons in PowerShell are used to determine "True" versus "False". For example, in the non-PowerShell world:

```
4 > 10 is False  
10 = 5 is False  
15 ≤ 15 is True
```

- PowerShell defines two special objects: \$True, \$False.
  - PowerShell will execute comparisons from the command line:

```
4 -gt 10  
False  
  
10 -eq 5  
False  
  
15 -cle 15  
True
```

### Key Points

Any comparison, either of two or more objects or two or more property values, is designed to result in a True or False value. These comparisons are often referred to as *Boolean comparisons*, because they can only result in one of the two Boolean values, True or False.

Comparing two simple values or objects is a common enough task: You might compare two computer names to see if they are equal, or compare a performance counter value to a predetermined threshold value to see which of the two is greater.

Comparison operators sit between the two items you want to compare. You probably remember simple comparisons from grade school math:

```
4 > 10  
10 = 5  
15 ≤ 15
```

Windows PowerShell performs comparisons in substantially the same way, although it does not use the traditional mathematical symbols for comparisons.

Windows PowerShell defines two special objects, \$True, and \$False, which represent the Boolean values True and False. The comparison *4 is greater than 10*, for example, will produce \$False as its result, while *10 is equal to 10* would produce \$True.

Windows PowerShell enables you to execute comparisons right on the command-line. Simply type your comparison and press RETURN to see the result of the comparison.

## What Are the Comparison Operators?

- Main comparison operators:
  - -eq : Equal to
  - -ne : Not equal to
  - -le : Less than or equal to
  - -ge : Greater than or equal to
  - -gt : Greater than
  - -lt : Less than
- Case-insensitive operators (not commonly used)
  - -ieq : Equal to
  - -ine : Not equal to
  - -ile : Less than or equal to
  - -ige : Greater than or equal to
  - -igt : Greater than
  - -ilt : Less than
- Case-sensitive operators:
  - -ceq : Equal to
  - -cne : Not equal to
  - -cle : Less than or equal to
  - -cge : Greater than or equal to
  - -cgt : Greater than
  - -clt : Less than

### Key Points

Because the shell uses < and > for other purposes (>, for example, is used to redirect output to a file), those symbols are not available for use as comparison operators. Instead, the shell uses operators that come from other command-line shells, notably popular shells used in the Unix operating system.

The main comparison operators are:

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

When used to compare strings of characters (something you would commonly do with -eq or -ne), these operators are case-insensitive. That means the result of this command:

```
"Hello" -eq "HELLO"
```

would be True. If you need to make a case-sensitive string comparison, there is a complete set of case-sensitive operators:

- -ceq : Equal to (case-sensitive)
- -cne : Not equal to (case-sensitive)
- -cle : Less than or equal to (case-sensitive)
- -cge : Greater than or equal to (case-sensitive)

MCT USE ONLY. STUDENT USE PROHIBITED

- -cgt : Greater than (case-sensitive)
- -clt : Less than (case-sensitive)

The shell also has explicit case-insensitive operators:

- -ieq : Equal to (case-insensitive)
- -ine : Not equal to (case-insensitive)
- -ile : Less than or equal to (case-insensitive)
- -ige : Greater than or equal to (case-insensitive)
- -igt : Greater than (case-insensitive)
- -ilt : Less than (case-insensitive)

These case-insensitive operators are not commonly used, because they duplicate the behavior of the normal operators.

## Boolean Operators

- The Boolean operators (-and, -or, -not) are used in more-complex comparisons, working with complete sub-comparisons on each side

- and returns \$True when both sides are true
- or returns \$True when either or both sides are true
- not reverses True and False

```
4 -gt 10 -or 10 -gt 4  
4 -gt 10 -and "Hello" -ne "Hello"
```

```
(4 -gt 10 -and (10 -lt 4 -or 10 -gt 5) -and 10 -le 10)
```

```
(-not 5 -eq 5) -and (10 -eq 10)
```

- Parenthesis group sub-comparisons and define an order of execution

- The most deeply-nested parenthesis are executed first, and the execution is completed from left to right

### Key Points

When you need to perform a more complex comparison, you can use two special Boolean operators, -and, and -or.

These operators each work with a complete sub-comparison on either side. The -and operator will return True if both sub-comparisons are also True; the -or operator will return True if either or both of the sub-comparisons are True. Here are some examples:

```
4 -gt 10 -or 10 -gt 4          # returns True  
4 -lt 10 -and "Hello" -ne "hello" # returns False
```

You can use parentheses to group sub-comparisons and define an order of execution. Normally, comparisons are performed left-to-right, but when you use parentheses, the most deeply-nested parentheses are executed first. For example:

```
(4 -gt 10 -and (10 -lt 4 -or 10 -gt 5)) -and 10 -le 10
```

The innermost comparison is evaluated first, from left to right: Because 10 is not less than 4, that is False. Because 10 is greater than 5, that is True. Here is the result:

```
(4 -gt 10 -and (False -or True)) -and 10 -le 10
```

Because those two sub-comparisons are joined by the -or operator, the result is True, yielding this:

```
(4 -gt 10 -and True) -and 10 -le 10
```

The next comparison is False because 4 is not greater than 10, resulting in this:

```
(False -and True) -and 10 -le 10
```

The `-and` operator only returns True when both of its sub-comparisons are True. That isn't the case for the remaining parenthetical comparison, so our comparison now looks like this:

```
False -and 10 -le 10
```

10 is less than or equal to 10, resulting in this:

```
False -and True
```

Once again, the `-and` operator will return False because one of its sub-comparisons is False. So the final result of the comparison is False:

```
PS C:> (4 -gt 10 -and (10 -lt 4 -or 10 -gt 5)) -and 10 -le 10  
False
```

A third Boolean operator is `-not`. This operator simply reverses True and False. Consider this comparison:

```
(-not 5 -eq 5) -and (10 -eq 10)
```

5 does equal 5, so that would normally be True, but the `-not` operator reverses it to False. 10 does equal 10, so that side is True. The final result, however, is False, because the first sub-comparison has been made False.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Basic Comparison Operators

- Use basic comparison operators in the shell



### Key Points

In this demonstration, you will learn how to use basic comparison operators within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Working with Basic Comparison Operators.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Understanding Pipeline Filtering

- The Where-Object cmdlet...
  - Is used to remove some objects from the pipeline
  - Removes objects based on specified criteria
  - Passes remaining objects to cmdlets further down the pipeline
- Where-Object uses the special placeholder variable `$_.Status` to signify the current pipeline object for comparison

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

- Caution: Using Where-Object requires processing each value in the cmdlet's result, which can be computationally expensive
  - Use cmdlet filtering before Where-Object where possible

### Key Points

Filtering is the process of removing some objects from the pipeline, usually because they don't meet some criteria that you have specified. You typically filter objects before passing the remaining objects on to another cmdlet for further actions.

Filtering is accomplished by means of the Where-Object cmdlet, or its predefined alias, Where. The main parameter of Where-Object is a script block, where you specify the criteria that an object must meet in order to remain in the pipeline. Objects that do not meet your criteria are removed from the pipeline.

Within this script block, the shell looks for the special placeholder variable, `$_.Status`. The shell replaces this variable with the current pipeline object. Essentially, the script block and whatever comparison you put inside of it will be examined once for each object that is piped into the cmdlet. Here is an example:

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Notice that the script block is enclosed within curly braces. If the Get-Service cmdlet generated 100 service objects, then the Where-Object comparison would be examined 100 times. Each time, the `$_.Status` variable would be replaced with a new service object. The `$_.Status` is followed by a period, which tells the shell that you are about to refer to one of the object's members – in this case, the Status property. This example is comparing the contents of each service's Status property to the text string *Running*. If the Status property contains *RUNNING*, *Running*, *running*, or some other string—keeping in mind that the `-eq` operator is case-insensitive—then that service will be piped out of Where-Object and to the next cmdlet in the pipeline. If the Status property of the service does not meet this criteria, the object is simply dropped.

**Note:** It can be easy to confuse the purpose of Select-Object and Where-Object. Remember that Where-Object is used to filter entire objects out of the pipeline; Select-Object is used to choose a subset of objects (using `-first` or `-last`), or to specify the properties of an object that you want to see.

If you are familiar with the Structured Query Language (SQL) used by most database products, then the cmdlet names for Select-Object and Where-Object should seem familiar – and the cmdlets perform the function you might expect them to. Select-Object is similar to the SQL SELECT statement, and is used to specify property names or an object subset; Where-Object is similar to the SQL WHERE clause, which specifies filtering criteria.

### **Where-Object Is Not Always the Best Choice**

In order to use Where-Object, you typically start with a Get- cmdlet, such as Get-Process or Get-ADUser. You pipe all of those objects to Where-Object, and it drops the objects that do not meet your criteria. For large sets of objects, Where-Object can be inefficient. In many cases, Get- cmdlets will offer their own filtering capabilities. When a cmdlet offers its own filtering capability (commonly through a –filter parameter), you should always use that in preference to Where-Object. Cmdlet filtering typically takes place at the data source, meaning the Get- cmdlet only has to retrieve objects that meet your criteria. That saves time and processing power.

When working with Active Directory cmdlets like Get-ADUser, in fact, the –filter parameter is mandatory, because accidentally querying all users from the domain could impose an unacceptable performance burden on a domain controller.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Understanding Pipeline Filtering

- Use Where-Object in the shell



### Key Points

In this demonstration, you will learn how to use Where-Object within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Understanding Pipeline Filtering.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Understanding Object Enumeration

- The `ForEach-Object` cmdlet...
  - Provides a way to perform an action against a set of objects
  - As with `Where-Object`, uses `$_` as a placeholder for the current piped-in object
  - Typically uses a script block to contain the action

```
Get-Service | Where-Object { $_.Status -eq "Stopped" }  
| ForEach-Object { $_.Start() }
```

- Remember: `ForEach-Object` is computationally expensive
  - Consider using `ForEach-Object` only when an existing cmdlet cannot accomplish your task on its own against a batch of objects

### Key Points

Many of the shell's so-called *action cmdlets*, those cmdlets which take some action or perform a task, are designed to work with entire collections of objects. For example, consider—but please do not actually run—this command:

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

This command retrieves all of the processes and then attempts to stop them all, something that would probably cause your computer to shut down unexpectedly.

Sometimes, however, you don't want to, or can't, work with objects in batches that way. In those cases, you may need to perform some action or operation against each object individually, one at a time.

**Note:** In Windows PowerShell v1, it was more common to have to enumerate objects to perform some action with them one at a time. In v2, Microsoft has added many more cmdlets that can work with batches of objects, reducing the number of scenarios where object enumeration is required.

Object enumeration is accomplished with the `ForEach-Object` cmdlet, which has built-in aliases `ForEach` and `%` (yes, the percent sign punctuation mark is an alias for `ForEach-Object`). The most common parameter you provide to `ForEach-Object` is a script block containing the action you want to be executed against each object that is piped in. Within that script block, the shell looks for the `$_` placeholder variable, and replaces it with the current piped-in object.

For example, here is a very simplistic example that retrieves all services, removes those that are running, and attempts to execute the `Start()` method of the remaining services:

```
Get-Service | Where-Object { $_.Status -eq "Stopped" } |  
ForEach-Object { $_.Start() }
```

In reality, this same task could be accomplished more easily in this fashion:

```
Get-Service | Where-Object { $_.Status -eq "Stopped" } | Start-Service
```

The Start-Service cmdlet can work with batches of input objects, so there is really no need to enumerate them and individually run their Start() methods.

**Note:** As a general rule, ForEach-Object becomes less and less necessary as Microsoft improves Windows PowerShell and releases additional cmdlets. If you find yourself using ForEach-Object, consider spending some time to see if there is a way to accomplish your task more efficiently by using a cmdlet that can work with a batch of objects. ForEach-Object is still valuable to know for those situations where a batch-processing cmdlet is not available.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Object Enumeration

- Use `ForEach-Object` in the shell



### Key Points

In this demonstration, you will learn how to use `ForEach-Object` within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Object Enumeration.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Filtering and Enumerating Objects in the Pipeline

- Exercise 1: Comparing Numbers (Integer Objects)
- Exercise 2: Comparing String Objects
- Exercise 3: Retrieving Processes from a Computer
- Exercise 4: Retrieving Services from a Computer
- Exercise 5: Iterating Through a List of Objects

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are a system administrator. One of your tasks in the morning as your shift is starting is to check the status of the processes and services for the servers you are responsible for. Lately, you have a particular service installed on several computers that have been having issues. Your corporate monitoring solution has not always been alerting that the service has stopped running.

#### Lab Setup

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

1. Start the **LON-DC1** virtual machine. You do not need to log on, but wait until the boot process is complete.
2. Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Start the **LON-SVR2** virtual machine. You do not need to log on at this time.
4. Open a Windows PowerShell session as **Administrator**.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 1: Comparing Numbers (Integer Objects)

### Scenario

You need to check the status of processes and services on your server; however, before you begin, verify that PowerShell's comparison capabilities can indeed compare simple and complex expressions.

**Note:** An *integer object* is any whole number, such as 4 or 5. Numbers that have a fraction, such as 2.3, are not integers. Windows PowerShell can sometimes interpret a string as a number, when the number is enclosed in quotation marks, such as "7." However, integers are not normally enclosed in quotation marks.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Compare using a simple expression.
2. Compare more complex expressions.

► **Task 1: Compare using a simple expression**

- Open **Windows PowerShell** and create an expression that determines whether 3 is greater than 5.

► **Task 2: Compare using a more complex expression**

- Create a complex expression that determines whether 3 is smaller than 4 and 5 is equal to 5.

**Results:** After this exercise, you will have determined that 3 is smaller than 5 by having True returned, and also that the complex expression that compares whether 3 is smaller than 4 and 5 is equal to 5 resolves to True because both sides of the complex expression alone resolve to True.

## Exercise 2: Comparing String Objects

### Scenario

Continuing with the previous exercise, you want to expand the comparison to see how string objects compare in PowerShell. In this exercise, look at both simple as well as complex expressions.

**Note:** A *string object* is any sequence of characters that is enclosed within single or double quotation marks, such as "powershell" or 'windows.'

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Compare using a simple expression.
2. Compare using a more complex expression.

► **Task 1: Compare using a simple expression**

- Create an expression that determines whether the string *PowerShell* is equal to *powershell*.

► **Task 2: Compare using a more complex expression**

- Create a complex expression that determines whether the string *logfile* is equal to *logfiles* and *host* is equal to *HOST*. In the latter string, use a case-sensitive expression.

**Results:** After this exercise, you will have determined that *PowerShell* is considered equal to *powershell* using the default comparison operators, and that the complex expression comparing *logfile* to *logfiles* and *host* to *HOST* resolves to False because one side of the complex expression alone resolves to False.

## Exercise 3: Retrieving Processes from a Computer

### Scenario

With the previous exercises complete, you're ready to try out PowerShell with a few administrative activities. You manage a large number of servers and would like to use PowerShell to easily query other servers to retrieve a list of the running processes.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Retrieve a list of processes.
2. Retrieve objects from a remote computer.

► **Task 1: Retrieve a list of processes**

- Retrieve a list of running processes.

► **Task 2: Retrieve objects from a remote computer**

- From LON-SVR1, retrieve a list of running processes on LON-DC1.

**Results:** After this exercise, you will have retrieved the list of processes running remotely on LON-DC1 from LON-SVR1.

## Exercise 4: Retrieving Services from a Computer

### Scenario

You would like to use PowerShell to easily query other servers to retrieve a list of the services installed.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Retrieve a list of services.
2. Retrieve objects from a remote computer.
3. Filter a collection of objects.

► **Task 1: Retrieve a list of services**

- Retrieve a list of services.

► **Task 2: Retrieve objects from a remote computer**

- From LON-SVR1, retrieve a list of services from LON-DC1.

► **Task 3: Filter a collection of objects**

- From LON-SVR1, retrieve a list of services from LON-DC1 that have a name that starts with the letter "w."

**Results:** After this exercise, you will have retrieved a list of all the installed services that start with the letter *w* on LON-DC1 from LON-SVR1.

## Exercise 5: Iterating Through a List of Objects

### Scenario

You have a long list of systems that you administer that includes desktops, application servers, and Active Directory servers. Often, you have to perform actions only on the application servers so you need a method to be able to filter those out from the system list.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Create a file.
2. Import the contents of a file.
3. Iterate through a list of objects.
4. Filter a list of objects.

#### ► Task 1: Create a file

- Create a txt file **C:\Systems.txt** with the below entries. You can do this from the powershell command line if you are able or you can open Notepad and just enter the below data, each on a single line if you need.
  - **LON-SVR1.contoso.com**
  - **LON-SVR2.contoso.com**
  - **LON-DC1.contoso.com**
  - **LON-CLI1.contoso.com**

#### ► Task 2: Import the contents of a file

- Retrieve the contents of the file created.

#### ► Task 3: Iterate through a list of objects

- Retrieve the contents of the created file and display all the entries in lowercase.
- Start by running the command **Get-Content –Path "C:\Systems.txt" | Get-Member**. Notice that the **String** objects each have several methods. To execute one of those methods, you will reference the **String** object (in a **ForEach-Object** script block—do so by using the `$_.Placeholder`), type a period, then type the method name, followed by parentheses. For example, `$_ToUpper()`.

#### ► Task 4: Filter a list of objects

- Retrieve the contents of the file created and display only the entries where the name includes **SVR**.

**Results:** After this exercise, you will have retrieved a list of entries from a file, have changed all the characters to lowercase, and have also filtered the list of entries to return only the ones with the string **SVR** in the entry name.

## Lab Review

1. What if you wanted to filter a list of processes that started with s\* and with e\*? Would you use Where-Object or –Name?
2. When working with remote services, are there any advantages in using the –Name parameter versus using Where-Object?
3. What might interfere with doing any kind of query of a remote computer?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 3

# Advanced Pipeline Techniques

- Explain the difference between pipeline input `ByValue` and `ByPropertyName`
- Pipe objects to a cmdlet so that input objects and properties bind to a specified parameter of the cmdlet
- Modify objects in the pipeline to attach a new property having the same value as an existing property to facilitate pipeline parameter binding `ByPropertyName`
- Explain the use of the `-passthru` cmdlet parameter



A major difference between Windows PowerShell and older administrative scripting languages is that Windows PowerShell is designed first and foremost to facilitate the automation of administrative tasks. While the shell does support a powerful scripting language of sorts, it also enables administrators to accomplish a lot with much simpler and more familiar commands. It is possible to automate complex and detailed tasks in Windows PowerShell without writing something that you might recognize as a script.

In order to do that, however, you need to become skilled at leveraging Windows PowerShell's various command-line features. The most important of these features is the pipeline, and its ability to pass objects from one cmdlet to another. The way that cmdlets can be constructed enables them to perform a great deal of work with relatively little input – meaning less typing for you.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the difference between pipeline input `ByValue` and `ByPropertyName`.
- Pipe objects to a cmdlet so that input objects and properties bind to a specified parameter of the cmdlet.
- Modify objects in the pipeline to attach a new property having the same value as an existing property to facilitate pipeline parameter binding `ByPropertyName`.
- Explain the use of the `-passthru` cmdlet parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

## Parameters: The Key to the Pipeline

- Most PowerShell cmdlets support a variety of parameters
- Each parameter is designed to accept a very specific type of input
  - Some parameters work with integers, others with text, etc.
  - Some commands require specifying parameter names at the command line, while others are optional
  - Some parameters are optional, while others are mandatory
- Always remember: Consult the cmdlet's Get-Help for information about each parameter
  - Get-Help will show which are optional, as well as what type of input is expected

### Key Points

Most Windows PowerShell cmdlets support a variety of parameters, and each parameter is designed to accept a very specific type of input.



Follow along with the next few steps and pay close attention to the details called out in the text.

For example, close any copies of Windows Notepad that you may have running, and then open a new, fresh instance of Windows Notepad. Run this command:

```
Get-Process
```

Notice that your instance of Windows Notepad is listed. Try to stop that process by running this command:

```
Stop-Process notepad
```

Why did you get an error? The error message states that it was unable to bind the ID parameter, because it could not convert our input, *notepad*, to a 32-bit integer. Now examine the help for Stop-Process:

```
Help Stop-Process
```

Notice that there are various *parameter sets*. A parameter set is a grouping of parameters that can be used together. Some parameters can exist in several different sets. For example, suppose parameter set A contains parameters

–computername and –id. Parameter set B contains parameters –computername, –name, and –noclobber. That means you cannot use both the –id parameter and the –name parameter at the same time, because they are from different parameter sets. You can, however, use –id together with –computername, because those two exist in the same parameter set.

In the first parameter set of Stop-Process, the ID parameter is not optional, meaning you must provide a value for it. However, the actual name of the parameter, -id, is optional—it is enclosed in square brackets. When we ran Stop-Process without a parameter, the shell assumed we were providing “notepad” as the value for the –id parameter. But the –id parameter requires an Int32—a 32-bit integer—and “notepad” cannot be converted to that data type.

The third parameter set can accept the process name as a string, but if you choose to use that technique, then the –name parameter is not optional. Try this:

```
Stop-Process -name notepad
```

That works, because the shell understood that *notepad* was intended as input to the –name parameter, not the –id parameter.

The idea here is that all parameters are slightly different, and they are all designed to accept a certain kind of data.

MCT USE ONLY. STUDENT USE PROHIBITED

## Positional Parameters

- Positional parameters do not require specifying their name at the command line
  - Their use is based on the **position** of the parameter in the input
  - They are used to make typing easier, reduce input
- Positional parameters are displayed in square brackets in the cmdlet's help
- Parameter names, those which are specified at the command line, can be used in any order
  - Using parameter names is useful for code clarity

**Stop-Process -id 53 #Executes correctly**  
**Stop-Process 53 #Executes correctly**

**Stop-Process [-Id] <Int32[]>**

### Key Points

When you review the help for a cmdlet and see something like this:

**Stop-Process [-Id] <Int32[]>**

You are looking at a *positional parameter*. That is, the parameter itself is required, but you do not have to type the name. You could either run:

**Stop-Process -id 53**

Or you could run this:

**Stop-Process 53**

As long as the correct data type is provided in the correct position—in this case, the first parameter position—the shell will know what to do. This is actually made clearer in the full help:

**Help Stop-Process -full**

Here, you can see that the parameter is positional, that it occupies position number 1, and that it is a mandatory parameter:

**-Id <Int32[]>**  
Specifies the process IDs of the processes to be stop  
type "get-process". The parameter name ("Id") is optio  
  
Required? true  
Position? 1  
Default value  
Accept pipeline input? true (ByPropertyName)  
Accept wildcard characters? False

Here is another example, from the help for Get-ChildItem:

```
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Exclude <string[]>]
```

In this example, the entire `-path` parameter is optional. You can see that the entire parameter, including its data type of `string[]`, is enclosed in square brackets. However, the `-path` parameter name itself is also in a separate set of square brackets. This tells you that the parameter is optional, and that if you choose to use it, you can omit the parameter name provided the value is passed in the first position. The `-filter` parameter is also optional and positional; the `-exclude` parameter is optional, but it is not positional. If you choose to use `-exclude`, you must specify the parameter name.

Positional parameters are intended to make typing easier. You can imagine how tedious it would be to have to specify `-path` every time you wanted a directory:

```
Dir -path c:\
```

By having `-path` be positional, you can type a much shorter and more familiar command:

```
Dir c:\
```

However, using parameter names has two advantages:

- It is much clearer to someone else what the command is intended to do, because the parameter names help explain what is happening.
- When you use parameter names, you can specify them in any order you like, so you do not have to worry about remembering the correct order.

For example, consider this command:

```
Add-Member NoteProperty ComputerName LON-DC1
```

This cmdlet has not yet been covered in this course, and you might be unsure of what it does. But if the same cmdlet were written using parameter names, you might be able to get a better idea of what the cmdlet was doing:

```
Add-Member -memberType NoteProperty -Name ComputerName -Value LON-DC1
```

You might still have to look up the cmdlet's help file, but having the parameter names makes it easier to figure out which piece of data is being used for what.

You have been using positional parameters through this entire course. `Dir`, of course, which is an alias for `Get-ChildItem`, is one example. `Where-Object` is another example. Consider this example:

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Here is the same command with the parameter names filled in:

```
Get-Service | Where-Object -FilterScript { $_.Status -eq "Running" }
```

## Pipeline Input ByValue

- Many parameters are designed to accept input from the pipeline. This process is called *binding*.
  - You have already done this as you pipeline commands to each other. The second command accepts the input of the first.

**Get-Service | Where-Object { \$\_.Status -eq "Running" }**

- Here, the Where-Object cmdlet uses a positional parameter named `-InputObject` to accept pipeline input *ByValue*.

**-InputObject <psobject>**  
Specifies the objects to be filtered. You can...

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

### Key Points

So far in this course, you have probably focused primary on cmdlets where you have specified parameters, either by position or by name. Many parameters, however, are designed to accept input from the pipeline. In fact, whenever you pipe objects from one cmdlet to another, the cmdlet you are piping to must attach, or *bind*, the incoming pipeline objects to one parameter or another.

Consider this example:

```
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Here, you are piping service objects to Where-Object. But what does Where-Object do with those service objects, exactly? Consider the help file for Where-Object:

**-InputObject <psobject>**  
Specifies the objects to be filtered. You can a

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

The cmdlet only supports two parameters, `-FilterScript` and `-InputObject`. The filter criteria is being passed to the `-FilterScript` parameter by position. What is being done with `-InputObject`?

The `-InputObject` parameter is *binding* to the pipeline input. You can see in the help file that this parameter does accept pipeline input *ByValue*. The type of input it expects is shown right next to the parameter name: `<psobject>`. That indicates any kind of object will be accepted. The parameter's help says, "Specifies the objects to be filtered," meaning that this parameter contains the objects that need to be filtered.

Here's how it works:

1. Get-Service executes, and produces a collection of service objects.
2. The service objects are piped to Where-Object.
3. Where-Object sees that something has been provided for the –FilterScript parameter.
4. Where-Object also sees that a bunch of service objects have been piped in. Where-Object looks to see if any of its parameters will accept that kind of object ByVal.
5. Where-Object sees that its own –InputObject parameter can accept that kind of object ByVal.

Many cmdlets are capable of binding pipeline input ByVal. Look at the full help for cmdlets such as Stop-Process, Stop-Service, Start-Service, Remove-ChildItem, and others.

Note that binding ByVal always takes place first. In a moment, you will learn about binding ByPropertyName. That binding only occurs if no parameter was able to bind the pipeline input ByVal.

MCT USE ONLY. STUDENT USE PROHIBITED

## Pipeline Input ByPropertyName

- `ByValue` binding is easy: Acceptable values are enough to validate incoming input.
- Other parameters bind `ByPropertyName`, which requires a matching property name. This can be more difficult.
  - As before, reference the cmdlet's help for details
- Consider the help content for `Get-Service`:

**-ComputerName <string[]>**  
Gets the services running on the specified computers...

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

### Key Points

This kind of pipeline input binding is somewhat more complicated. In this technique, the shell looks at the parameter's name, and looks to see if the input objects have a matching property name. If they do, then that matching property is *bound* to the parameter.

Note that this kind of binding only occurs if the shell was unable to bind the input `ByValue`. For example, examine the help for the `Invoke-Command` cmdlet. Notice that the `-inputObject` parameter binds pipeline input `ByValue`, and accepts objects of type `[object]`. That object type, `[object]`, is a generic type - *everything*, in fact, is an object. That means `-inputObject` will always bind all pipeline input `ByValue`. There will be no opportunity for a parameter to bind pipeline input `ByPropertyName`.

Examine the full help for `Get-Service` and you will notice the following parameter:

**-ComputerName <string[]>**  
Gets the services running on the specified computers

Type the NetBIOS name, an IP address, or a fully qualified name, a dot (.), or "localhost".

This parameter does not rely on Windows PowerShell being configured to run remote commands.

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

(This output was truncated to fit in this book; view the help for `Get-Service` in Windows PowerShell to read the complete text.)

This help explains that the `-computername` parameter accepts pipeline input `ByPropertyName`. One way to run this command would be:

```
Get-Service -computername LON-DC1
```

You could even provide a comma-separated list of computer names:

```
Get-Service -computername LON-DC1,SEA-DC1
```

However, another technique would be to pipe in objects that have a `ComputerName` property. For example, you could use Windows Notepad to create a file like this:

```
ComputerName  
LON-DC1  
SEA-DC1
```

You would save the file as a CSV file. Even though it does not contain any commas, it's technically still a valid CSV file: It has a header row with a column name, and it has two rows of data. It doesn't need commas, because it contains only one column. You could import the file into the shell by running something like this:

```
Import-Csv computernames.csv
```

The `Import-Csv` cmdlet would output two objects, each with a `ComputerName` property. Those objects could be piped to `Get-Service`:

```
Import-Csv computernames.csv | Get-Service
```

Because the `ComputerName` property of the objects match the `-computername` parameter name, your two computer names would be sent as input to the `-computername` parameter, and `Get-Service` would attempt to retrieve the services from both computers.

**Note:** Pipeline parameter binding `ByPropertyName` only works when the input objects have a property name that fully and exactly matches the parameter name (although the match is case-insensitive). A property named `comp` would not match the `-computername` parameter, for example.

## Renaming Properties

- Sometimes you need to bind cmdlets, but those cmdlets use different property names for the same data. What do you do?
  - `ByValue` is easy. Values are enough.
  - `ByPropertyName` requires renaming the property.
- Renaming properties uses `Select-Object`.
  - `Select-Object @{Label="NewName";Expression={$_._OldName}}`

```
Get-ADComputer -filter * | Select-Object  
@{Label="ComputerName";Expression={$_._Name}}  
  
Get-ADComputer -filter * | Select-Object *,  
@{Label="ComputerName";Expression={$_._Name}} |  
Get-Service
```

### Key Points

Suppose you run a cmdlet that generates computer names, and you want to pipe those computer names to a second cmdlet that has a `-computername` parameter. The problem is that the first cmdlet might not put the computers' names into a `computername` property. `Get-ADComputer` is an example of this.

**Note:** We have not yet covered the Active Directory cmdlets, but we will do so in an upcoming module. For now, we will use the `Get-ADComputer` cmdlet as an example.



You can follow along with the commands on the next pages if you are using your classroom domain controller and have either a Windows PowerShell console, or the Windows PowerShell ISE, open.

Run the following commands:

```
Import-Module ActiveDirectory  
Get-ADComputer -filter *
```

Your result will look something like this:

```
DistinguishedName : CN=SERVER-R2,OU=Domain  
Controllers,DC=company,DC=pri  
DNSHostName : server-r2.company.pri  
Enabled : True  
Name : SERVER-R2  
ObjectClass : computer  
ObjectGUID : 0f04c9d6-44af-4e4f-bb7e-4ebc52ab343f  
SamAccountName : SERVER-R2$  
SID : S-1-5-21-1842503001-3345498097-2301419571-1000  
UserPrincipalName :
```

These computer objects have a Name property. That will not match up to a -computerName parameter that is binding pipeline input ByPropertyName. You could not, for example, run this command successfully:

```
Get-ADComputer -filter * | Get-Service
```

You can do something similar, though. The trick is to use Select-Object to add a new property to your computer objects. The property will be named ComputerName, and it will contain whatever value is already in the existing Name property. This is the command:

```
Get-ADComputer -filter * | Select  
@{Label="ComputerName";Expression={$_.Name}}
```

The result will be objects that have only a ComputerName property. If you also wanted to retain the other existing properties of the computer object, you would just need to tell Select-Object to also include them in the output:

```
Get-ADComputer -filter * | Select *,  
@{Label="ComputerName";Expression={$_.Name}}
```

Adding the \* to the property list includes all properties from the input object. With a ComputerName property added, this command can now succeed (assuming you have permission to connect to the remote computers, of course):

```
Get-ADComputer -filter * | Select *,  
@{Label="ComputerName";Expression={$_.Name}} | Get-Service
```

This is a very powerful technique that helps reduce the need to write more complicated scripts.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Pipeline Input

- Use pipeline input parameter binding



### Key Points

In this demonstration, you will learn how to use pipeline input parameter binding.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Working with Pipeline Input.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Understanding Passthrough

- Some “action” cmdlets accept pipeline input, but do not provide output.

```
New-ADUser -name JohnD -samaccountname JohnD
```

# This command has no output. Results can't be piped to other commands as is.

- Cmdlets that provide no output by default require an extra parameter (-passThru) to pass on its objects.

```
New-ADUser -name JohnD -samaccountname JohnD  
-passThru | Enable-ADAccount
```

#New-ADUser result is now piped to Enable-ADAccount

- passThru is used by many cmdlets. See help for details.

### Key Points

Most so-called *action* cmdlets can accept pipeline input, but many do not produce any output at all. The New-ADUser cmdlet is a good example of this. For example, you can create a simple user account by running a command like this:

```
New-ADUser -name JohnD -samaccountname JohnD
```

However, the command will not produce any output, and the user will be created in a Disabled state. One option for enabling the user would be to run the command above, and to then run a command like this one:

```
Get-ADUser -filter "name -eq JohnD" | Enable-ADAccount
```

However, the New-ADUser cmdlet, as well as many other action cmdlets, supports a -passThru parameter. When used, this parameter tells the cmdlet to output whatever object it was working with or that it created. So you could run something like this:

```
New-ADUser -name JohnD -samaccountname JohnD -passThru |  
Enable-ADAccount
```

Stop-Service is another cmdlet that supports -passThru. Normally, Stop-Service produces no output. However, when you include -passThru, it does. You could run this:

```
Get-Service -name BITS | Stop-Service -passthru
```

The command would now generate output, showing you the services that it attempted to stop.

## Demonstration: Using the –Passthru Parameter

- Use the –passThru parameter of cmdlets that support it



### Key Points

In this demonstration, you will learn how to use the –passThru parameter of cmdlets that support it.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod03\Democode\Using the - Passthru Parameter.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab C: Using Pipeline Parameter Binding

- Exercise 1: Using Advanced Pipeline Features
- Exercise 2: Working with Multiple Computers
- Exercise 3: Stopping a List of Processes
- Exercise 4: Binding Properties to Parameters

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are a system administrator for a company with about 100 users and desktop computers. You want to perform a set of tasks against those users and computers using Windows PowerShell. You want to accomplish those tasks using data from external files. You need to add a set of users to Active Directory, using properly-formatted and improperly-formatted CSV files. You also need to regularly reboot a set of computers based on the contents of a CSV file. Finally, you also need to stop a list of processes on computers, based on the contents of a CSV file.

### Lab Setup

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

1. Start the **LON-DC1** virtual machine, and then logon by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Start the **LON-SVR1** virtual machine. You do not need to log on.
3. Start the **LON-SVR2** virtual machine. You do not need to log on.
4. On LON-DC1, open a Windows PowerShell session as **Administrator**.

## Exercise 1: Using Advanced Pipeline Features

### Scenario

You have created a properly-formatted CSV formatted file of new users that need to be added to your Active Directory installation. You want to use PowerShell to automate the process.

The main tasks for this exercise are as follows:

1. Import the Active Directory module.
2. Import a CSV file.
3. Create users in Active Directory using advanced pipeline features.

► **Task 1: Import the Active Directory module**

- On LON-DC1, import the **Active Directory** module.

► **Task 2: Import a CSV file**

- Import the CSV file **E:\Mod03\Labfiles\Module3\_LabC\_Exercise1.csv**.

► **Task 3: Create users in Active Directory**

- Using advanced pipeline features, import the CSV file **E:\Mod03\Labfiles\Module3\_LabC\_Exercise1.csv**, and create new Active Directory users.

**Results:** After this exercise, you will have automated the creation of users in the contoso.com domain.

## Exercise 2: Working with Multiple Computers

This is an *optional exercise* to complete if you have extra time, or if you want to explore on your own.

### Scenario

You manage several types of systems from clients, to servers, and domain controllers. You are allowed to update and reboot servers before 8AM. You have a list in a CSV file of all the systems in your company and only want to reboot the ones that are servers.

The main tasks for this exercise are carried out on LON-DC1 and are as follows:

1. Create a CSV file.
2. Import a CSV file and filter the results.
3. Restart a group of computers.

#### ► Task 1: Create a CSV file

- On LON-DC1, create a CSV file **C:\Inventory.csv** with the below contents. Again, you can do this from the powershell command line if you are able or you can open Notepad and just enter the below data, each on a single line if you need.
  - **Type,Name**
  - **AD,LON-DC1**
  - **Server,LON-SVR1**
  - **Server,LON-SVR2**
  - **Client,LON-CLI1**

#### ► Task 2: Import a CSV file and filter the results

- Import the CSV file created, and list only the systems where the Type is *Server*.

#### ► Task 3: Restart a group of computers

- Import the CSV file created, and restart the systems where the Type is *Server*.

**Note:** If someone is logged in to LON-SVR1 and LON-SVR2 you may receive an error message. To overcome the error message and achieve a successful re-start the virtual machines, log off from them and then re-run the script.

**Results:** After this exercise, you will have restarted a filtered group of servers.

## Exercise 3: Stopping a List of Processes

### Scenario

There are a set of inappropriate processes that are currently running on systems within your network. You have been given a CSV file that contains the list of computers and associated processes that must be stopped on each computer.

The main tasks for this exercise are as follows:

1. Start 2 applications.
2. Create a CSV file.
3. Import a CSV file and iterate through a collection of objects.
4. Stop a list of processes.

► **Task 1: Start two applications**

- On LON-DC1, open one session of Notepad and Wordpad.

► **Task 2: Create a CSV file**

- Open Windows PowerShell and create a CSV file named **C:\Process.csv** with the following entries.
  - **Computer,ProcName**
  - **LON-DC1,Notepad**
  - **LON-DC1,Wordpad**

► **Task 3: Import a CSV file and iterate through a collection of objects**

- Import the CSV file created in the previous task, **C:\Process.csv**, and select the **ProcName** property to display.

► **Task 4: Stop a list of processes**

- Import the CSV file created in Task 2, **C:\Process.csv**, and stop each of the processes listed.

**Results:** After this exercise, you will have stopped a list of processes based on values imported from a CSV file.

## Exercise 4: Binding Properties to Parameters

**Note:** Complete this exercise only if you have finished the previous exercises and still have time remaining. If you do not have time, you can complete this exercise on your own.

### Scenario

You were provided a CSV formatted file with improper formatting. That file includes a second set of new users that need to be added to your Active Directory installation. You want to use PowerShell to automate the process.

The main tasks for this exercise are as follows:

1. Import a CSV file.
2. Create users in Active Directory.

► **Task 1: Import a CSV file**

- On LON-DC1, import the CSV file **E:\Mod03\Labfiles\Module3\_LabC\_Exercise4.csv**.

► **Task 2: Create users in Active Directory**

- Import the CSV file **E:\Mod03\Labfiles\Module3\_LabC\_Exercise4.csv** and create new Active Directory users by specifically binding properties to parameters. From the CSV file, map the properties according to the following table:

CSV file property	New-ADUser parameter
EmployeeName	Name and SamAccountName
Town	City
Unit	Department
Id	EmployeeNumber

- Also, hard code the value for the **Company** parameter as **Contoso**.

**Hint:** You have already seen the syntax used to rename a property. You will have to repeat this syntax once for each property than you want to rename. To save typing, you may wish to complete this task in the Windows PowerShell ISE. That will allow you to type the syntax once, and then copy and paste it (and modify the values of each pasted copy) to help reduce the amount of typing.

**Results:** After this exercise, you will have automated the creation of 20 users in the contoso.com domain.

## Lab Review

1. Do all cmdlets accept pipeline input like New-ADUser?
2. What is the advantage of a cmdlet parameter that accepts pipeline input?
3. Before stopping a process, could a check be made to only stop the process if it was using more than a certain amount of physical or virtual memory?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- Where can the `$_` placeholder be used?
- What cmdlet do you use to remove selected objects from the pipeline?
- What kind of object is output by `Select-Object` when you specify property names?
- What kind of object is output by `Measure-Object`?

### Class Discussion

- Common issues related to core cmdlets
- Best practices related to core cmdlets



### Review Questions

1. Where can the `$_` placeholder be used?
2. What cmdlet is used to remove selected objects from the pipeline?
3. What kind of object is output by `Select-Object` when you specify property names?
4. What kind of object is output by `Measure-Object`?

### Common Issues Related to Core cmdlets

Identify the causes for the following common issues related to the core cmdlets and fill in the troubleshooting tips. For answers, refer to relevant lessons in the module.

Issue	Troubleshooting tip
You use <code>Select-Object</code> to filter objects out of the pipeline, but the cmdlet does not work.	
You use <code>\$_</code> at the command-line but it appears to be empty.	

### Best Practices Related to Core cmdlets

Supplement or modify the following best practices for your own work situations:

- Feel free to use cmdlet's aliases and shortened parameter names when typing on the command-line. However, when you begin saving your commands in scripts or other files for future use, spell out cmdlet names and parameter names to improve readability.
- Avoid using `ForEach-Object` if you can. There are scenarios where it is absolutely required, and there is no danger or performance impact associated with it, but in many cases you can accomplish the same

task with less effort when you select a cmdlet capable of accepting collections of objects and processing them in a batch.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 4

## Windows® Management Instrumentation

### Contents:

Lesson 1: Windows Management Instrumentation Overview	4-3
Lesson 2: Using Windows Management Instrumentation	4-22
Lab: Using Windows Management Instrumentation in Windows PowerShell	4-35

## Module Overview

- Explain the structure of the WMI repository
- Configure and inspect WMI security
- Retrieve WMI information using Windows PowerShell
- Manipulate WMI objects within the Windows PowerShell pipeline
- Locate and use WMI documentation on the Internet
- Specify WMI filtering criteria



Windows® Management Instrumentation, or WMI, is a management technology included with the Microsoft® Windows operating system. It has been available since Windows NT® 4.0, and it was designed to provide a more consistent way of accessing configuration and other information from both local and remote computers.

For many years, WMI has been one of administrators' most powerful and approachable tools for obtaining management information and making certain configuration changes, especially on remote computers. However, actually using WMI has not always been easy or straightforward; older technologies such as Microsoft Visual Basic® Scripting Edition (VBScript) required a programming-style approach to using WMI, and not all administrators have been comfortable with that approach.

Windows PowerShell® provides a powerful and flexible means of accessing WMI through straightforward commands. In many ways, Windows PowerShell may offer the easiest and most approachable means for administrators to work with WMI.

### Module Objectives

After completing this module, you will be able to:

- Explain the structure of the WMI repository.
- Configure and inspect WMI security.
- Retrieve WMI information using Windows PowerShell.
- Manipulate WMI objects within the Windows PowerShell pipeline.
- Locate and use WMI documentation on the Internet.
- Specify WMI filtering criteria.

## Lesson 1

# Windows Management Instrumentation Overview

- Explain the structure of the WMI repository
- Configure and inspect WMI security
- Locate and use WMI documentation on the Internet



It is important to understand that WMI is intended to be different things to different audiences. Software developers interact with WMI, as do applications such as Microsoft System Center Configuration Manager. This module will focus on WMI as an administrative tool and will deal with only those elements of WMI that offer value as a day-to-day administrative tool.

### **Lesson Objectives**

After completing this lesson, you will be able to:

- Explain the structure of the WMI repository.
- Configure and inspect WMI security.
- Locate and use WMI documentation on the Internet.

MCT USE ONLY. STUDENT USE PROHIBITED

## About WMI

- WMI is a management technology that was first implemented in Windows NT 4.0.
  - WMI seeks to present information in a standardized manner
  - WMI presents information in a manner consistent with the Common Information Model (CIM)
- Consistency does not equal standardized
  - Few hard-and fast rules for software developers
  - WMI is implemented differently across different Microsoft and third-party products
  - Many WMI implementations are not well-documented
- Many PowerShell cmdlets wrap around WMI functionality
  - More administrator-friendly and task-specific uses. Hides WMI's underlying complexities

**You will primarily use WMI for retrieving management information about Windows, hardware, and applications**

### Key Points

WMI is a management technology that has been implemented in the Windows operating system since Windows NT 4.0. Each new version of Windows, as well as each new version of many other Microsoft products, extends WMI, adding new features and capabilities to it.

WMI presents information in a manner consistent with the Common Information Model, or CIM, which was developed by the Distributed Management Task Force, or DMTF, an industry group that Microsoft and other vendors participate in.

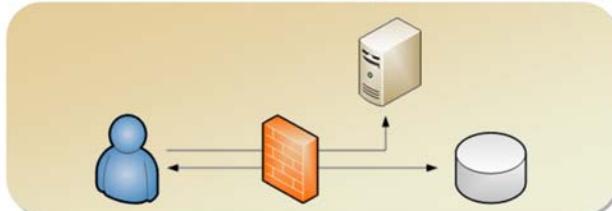
Although WMI seeks to present information in a standardized manner, it imposes few hard-and-fast rules that software developers must follow. As a result, WMI is often implemented very differently across various Microsoft and third-party products. That can, unfortunately, make WMI difficult to learn, and the problem is compounded by the fact that many WMI implementations are not well-documented anywhere. Windows PowerShell can make it easier to access WMI after you know which bit of WMI you want to get to; Windows PowerShell cannot, however, entirely make up for a lack of documentation or a lack of strong implementation standards.

Part of the complexity of WMI comes from its need to serve many different audiences, including software developers who often need a more detailed and low-level presentation of management information. Over the long term, you may see more and more Windows PowerShell cmdlets being written as wrappers around WMI functionality. Such wrappers would provide a more administrator-friendly and task-specific means of working with WMI, while hiding many of WMI's underlying complexities. In fact, some of the cmdlets you use today may be using WMI under the hood. But WMI is in little danger of going away completely. It's a valuable technology and, despite inconsistencies and other shortcomings, it offers powerful capabilities.

As an administrator, you will probably deal with WMI primarily to retrieve management information about Windows, computer hardware, and products running on Windows. In limited cases, you may also use WMI to make configuration changes.

## WMI Communications

- WMI communications take place using the Remote Procedure Call (RPC) protocol
  - Leverages an endpoint mapper on the remote computer
  - This endpoint mapper enables WMI communications to start over a well-known TCP port, then shift to a random TCP port for the remainder of the conversation
  - This randomness makes firewall configuration challenging



- The Windows Firewall supports a dynamic *Remote Management* exception
  - This exception will follow WMI's RPC traffic as it changes ports

### Key Points

When you use WMI to connect to a remote computer, you are essentially using the WMI subsystem on your local computer to make a connection to the WMI service on the remote computer. The communications between the two computers takes place using the Remote Procedure Call, or RPC, protocol.

RPC is an older, established protocol that has been in use in the Windows operating system for more than a decade. However, as local firewalls become more common, RPC has become more difficult to use and manage.

Typically, a computer's first RPC connection is to an *endpoint mapper* on the remote computer. This endpoint mapper is a portion of the RPC service on the remote computer, and it runs on a well-known TCP port. You might imagine that you could simply open that port in a local firewall, but that port does not actually carry the bulk of the RPC communications. Instead, the endpoint mapping process selects a new, random TCP port for the remainder of the conversation. Because that port is selected randomly, it is difficult to create static firewall rules to allow RPC traffic.

The Windows Firewall supports a dynamic "Remote Management" exception that can allow the RPC traffic needed for WMI. Other firewall software may have similar capabilities; check with the vendor documentation for specifics. Other configuration settings, including User Account Control (UAC) and Distributed Component Object Model (DCOM) can also affect WMI communications; the Web pages at the previously referenced URLs provide information on configuring these settings.

**Note:** For ease of use in the classroom environment, you may wish to disable the Windows Firewall entirely on your lab virtual machines. Although this might not be something you can do in production, it allows you to focus on using WMI in the lab, rather than on configuring firewalls and other Windows components.

## WMI Structure

- WMI is not a single piece of software.
  - Several different providers contribute its parts, each of which connects the WMI service to a given product, technology, feature, and so on.
  - Each WMI Provider gives the WMI Service access to various portions of Windows and other products.
- Information made available by a provider is registered into the WMI repository.
  - The repository is a centralized configuration database that tells WMI what information is available.
- The repository is organized into namespaces, which roughly correspond to products and technologies. These can be different based on the computer's configuration.

```
graph TD; Root["Root\\"] --- Cimv2["Cimv2"]; Root --- MicrosoftDNS["MicrosoftDNS"]; Root --- MicrosoftActiveDirectory["MicrosoftActiveDirectory"]; Root --- SecurityCenter["SecurityCenter"];
```

### Key Points

Behind the scenes, WMI is not a single piece of software. It includes several different providers, each of which connects the WMI service to a given product, technology, feature, and so on. A WMI provider acts almost as a kind of device driver, giving the WMI service access to various portions of Windows and other products. For example, on Windows Server computers that have the Windows DNS service installed, a WMI provider enables the WMI service to query and create DNS records as well as DNS configuration settings.

All of the information that a provider makes available is registered in the *WMI repository*, a centralized configuration database that tells WMI what information is available to it on that particular computer. It is important to understand that the information you get from WMI is not actually kept in the repository; the repository maintains a list of *what information is available*, and WMI actually retrieves that information dynamically when it is requested.

The repository is organized into namespaces, which roughly correspond to products and specific technologies. The namespaces are hierarchical, which means they can have sub-namespaces. The top-level namespace is *root*; other namespaces might include:

- Root\Cimv2
- Root\MicrosoftDNS
- Root\MicrosoftActiveDirectory
- Root\SecurityCenter

Not every Windows computer contains the same namespaces. For example, a client computer would not contain Root\MicrosoftActiveDirectory, and a server computer would not contain Root\SecurityCenter.

**Note:** Root\Cimv2 is the default namespace. When you are working with WMI, if you do not explicitly state a namespace to use, WMI will use Root\Cimv2.

You do not need a namespace installed on your computer to work with that same namespace on a remote computer. For example, if you are using a Windows 7 computer and you want to work with Windows DNS on a remote server, you can do so through the remote computer's Root\MicrosoftDNS namespace—even though that namespace does not exist on your client computer.

MCT USE ONLY. STUDENT USE PROHIBITED

## Classes

- Within each WMI namespace are one or more classes. A class is an abstract definition of a manageable component.
- WMI classes generally relate to operating system and hardware components, containing properties and methods.
  - Properties: Provide access to management configuration information. All classes have at least one property.
  - Methods: Cause some action to be taken. Not in every class.



Win32\_Account  
Win32\_BIOS  
Win32/Desktop  
Win32\_Fan  
Win32\_Group  
Win32\_Keyboard  
Win32\_LogicalDisk  
**Win32\_NetworkAdapterConfiguration**  
Win32\_NTDomain  
Win32\_Product  
Win32\_Service

### Key Points

Within each namespace, WMI defines one or more *classes*. A class is an abstract definition of a manageable component. The Root\Cimv2 namespace, for example, contains a class named Win32\_TapeDrive. This class defines the attributes of a tape drive, and it exists in the namespace whether the computer has a tape drive attached or not.

**Note:** The Root\Cimv2 namespace class names are almost all prefixed by Win32\_, even on a 64-bit computer. This prefix is not used on classes in other namespaces. In fact, few other namespaces use any kind of class name prefix at all.

Other classes from the Root\Cimv2 namespace include:

- Win32\_Account
- Win32\_BIOS
- Win32/Desktop
- Win32\_Fan
- Win32\_Group
- Win32\_Keyboard
- Win32\_LogicalDisk
- Win32\_NetworkAdapterConfiguration
- Win32\_NTDomain
- Win32\_Product
- Win32\_Service

As you can see, the Root\Cimv2 namespace contains the classes related to the Windows operating system and to the computer's hardware. It is one of the few namespaces included on every Windows computer, although the classes within the namespace differ between server and client operating systems and between different versions of Windows.

Classes define two important elements: Properties and methods. All classes have at least one property, and like the other objects in Windows PowerShell the properties of a class provide access to management and configuration information. Some classes have methods, which cause some action to be taken, such as restarting a computer or making a configuration change.

MCT USE ONLY. STUDENT USE PROHIBITED

## Instances

- Instances are real-world occurrences of a class

**If your computer has four logical disks,  
you have four instances of Win32\_LogicalDisk**

- Instances contain the properties and methods that are defined by their class
  - Many instances are read-only, allowing retrieval of data only
  - Other instances can be updated, allowing data to be changed

**Namespaces, classes, and instances are often very  
different. Some sleuthing is often required to find the data  
or methods that you want**

## Key Points

A real-world occurrence of a class is called an *instance*. For example, if your computer has four logical disks, you have four instances of the Win32\_LogicalDisk class. If your computer does not have an attached tape drive, it has zero instances of the Win32\_TapeDrive class—although the class itself still exists as an abstract definition.

Instances are objects just like the others you have used in Windows PowerShell. Instances have whatever properties and methods that are defined by their class, and you can work with these properties and methods within the shell's pipeline.

In many cases, instances are read-only, meaning that you can retrieve a property's values but you cannot directly change a property's values. This is especially true of the classes in the Root\Cimv2 namespace. If you want to change a configuration setting, the class must usually define a method, which you invoke to make changes.

Outside the Root\Cimv2 namespace, things can sometimes work differently. In some cases, you can change a property's value by assigning a new value to it. This is the case within the WMI namespace for Internet Information Services (IIS) 6, for example. This is one of the major inconsistencies in WMI: At different times, different product teams within Microsoft have had different ideas about how WMI might be used to help administer their product or technology or feature. In some cases, one product team's ideas were adopted by other product teams, creating some consistency and similarity between different namespaces; in other cases, a product team's ideas were never reused elsewhere, making that product team's namespaces unique within WMI.

## Finding the Class You Want

- Finding the right class involves research
  - There is no central repository of WMI classes
  - There are few tools that enable searching for WMI classes based on keywords
  - PowerShell can display a list of classes:

```
Get-WmiObject -list  
Get-WmiObject -list -class *user*
```

- Three techniques for finding your class (and data):
  - **Experience:** Previous experience, educated guesses, and Internet Web sites that document class information
  - **Exploration:** WMI “explorer” or “browser” tools
  - **Examples:** Using Internet search engines to search for task-related keywords will assist in finding examples

### Key Points

With tens of thousands of WMI classes in existence, finding the one that can help you accomplish a particular task can be very difficult. There is no central directory of WMI classes, and there are few tools (apart from Internet search engines, of course) that allow you to search for WMI classes based on keywords.

You must rely on three basic techniques to find the class you want:

- **Experience.** As you become more experienced in using WMI, you may be able to make educated guesses about a class name, and then use tools or search engines to validate your guess. Asking other experienced administrators can help, too. Several online communities provide ways to ask other administrators, including:
  - The “Windows PowerShell” Internet newsgroup (accessible at <http://go.microsoft.com/fwlink/?LinkId=193516>)
  - [www.PoshComm.org](http://www.PoshComm.org)
  - [www.PowerShell.com](http://www.PowerShell.com)
  - [www.ScriptingAnswers.com](http://www.ScriptingAnswers.com)
  - [www.ConcentratedTech.com](http://www.ConcentratedTech.com)
- **Exploration.** Various WMI “explorer” or “browser” tools allow you to browse the WMI repository of a local or remote computer. These typically present classes in alphabetical order, making it easier to locate classes by name and to see what classes are available.
- **Examples.** By using Internet search engines to search for task-related keywords (such as “WMI BIOS” or “WMI disk space”), you can often find examples that others have written, and those examples—even if they do not do exactly what you need—can help you discover the WMI class or classes related to your task. The Microsoft TechNet Script Center Repository

(<http://go.microsoft.com/fwlink/?LinkId=193517>) is also a good starting point for examples related to a wide variety of administrative tasks.

**Note:** Notice that even examples in other technologies, such as VBScript, JScript, or Perl, can be useful for discovering WMI class names. Don't entirely discount an example simply because it is not specific to Windows PowerShell.

Also note that you can ask Windows PowerShell to display a list of classes:

```
Get-WmiObject -list
```

This command lists the classes in the default namespace; to list classes in another namespace, add the `-namespace` parameter and specify the namespace name, such as `Root\SecurityCenter`.

The shell can also help you find classes that have a particular string of characters in the class name:

```
Get-WmiObject -list -class *user*
```

Because the `-class` parameter is positional, you can even omit the `-class` parameter name:

```
Get-WmiObject -list *user*
```

## Be Aware of Changes

- WMI is constantly evolving, constantly changing as products and operating systems change
- This means that a particular WMI class may behave slightly differently on different versions of Windows
  - Greater or fewer properties
  - Presence or absence of methods
  - Presence of absence of the entire class itself
- Namespaces can be different as well
  - Client and server computers typically have different namespaces
  - Namespaces are created when the product they serve is installed to the individual computer

### Key Points

WMI has evolved over the years, and it continues to do so. That means a particular WMI class may behave slightly differently in different versions of Windows, might have fewer properties in an older version of Windows, or might be missing entirely in an older version of Windows. Be very aware of these possibilities, especially when connecting to WMI on a remote computer.

Other than the Root\Cimv2 namespace, the namespaces available on each computer vary significantly, too. Client and server computers typically have different namespaces, and namespaces are often available only when their corresponding product or feature is installed.

## Documentation

- There is no central documentation repository for every WMI namespace and class
  - Core operating system classes in Root\Cimv2 are consistently documented
  - This documentation is available on the Internet within the Microsoft Developer Network (MSDN) library
  - Searching the library for your needed class is an effective way to locate the properties and methods you need

**Pay close attention to the product or Windows version associated with the class**

### Key Points

It would be nice if there was a central documentation repository for every WMI namespace and class. Unfortunately, there is not.

The core operating system classes—those in the Root\Cimv2 namespace—are consistently documented, and that documentation is available on the Internet as part of the Microsoft Developer Network (MSDN®) Library. Because the Library is periodically reorganized, changing its URLs, the easiest way to find the documentation for a class is usually to search for the class name in a search engine. For example, if you search for "Win32\_Service" you usually find that the first result takes you into the documentation page within the Library; from there, you can explore other classes in the same namespace by using the table of contents tree.

Outside the Root\Cimv2 namespace, documentation is much less consistent both in form and in availability. Some product teams have never been given the time to create documentation. In other cases, a product team may have created WMI classes for their product's own use and never intended those classes to be used by someone else, and so any documentation on those classes is not released to the public.

When documentation is available—and this is especially true of the classes in the Root\Cimv2 namespace—pay close attention to details that indicate which product or Windows version the class works with. Always keep in mind, however, that classes can and do differ between versions. Although the Win32\_Processor class is present on Windows NT 4.0, for example, the class behaves very differently from the version on Windows Server 2008 R2 (although in this example, the documentation page explains that difference).

## Demonstration: Finding and Using the Documentation

- Locate and interpret WMI class documentation on the Internet



### Key Points

In this demonstration, you will learn how to locate and interpret WMI class documentation on the Internet.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod04\Democode\Finding and Using the Documentation.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## WMI Security

- WMI security can be defined on an entire namespace or down to an individual class attribute.
  - By default, much of WMI's security is configured on the Root namespace. Other namespaces and classes inherit that root permission.
  - WMI security tends to allow nonadministrative users access to read local information.
  - Administrative users are permitted to remotely query information.
- WMI security is viewed and modified using the WMI Control MMC snap-in.
- WMI is sensitive to User Account Control.

**Important: Default WMI settings are appropriate for most situations.**

**It is not a good idea to manipulate default WMI security except for very specific reasons.**

### Key Points

WMI contains a robust security system. Security can be defined on an entire namespace, or it can be defined all the way down to an individual class attribute.

By default, much of WMI's security is configured on the Root namespace, with other namespaces, and their classes, inheriting that root permission.

Also by default, WMI security is configured to allow local users to query almost any information from WMI. Members of the local Administrators group (which in a domain environment usually includes the domain-level Domain Admins group) are permitted to query information remotely.

It is *not* a good idea to reconfigure WMI security unless you are aware of what you are doing and understand the full consequences. Modifying security can make WMI stop working altogether, or it can negatively affect products that rely on WMI—such as System Center Configuration Manager. The security defaults are good for most production environment, and it is often wisest to leave them alone.

WMI security can be viewed and modified by using the WMI Control snap-in in the Microsoft Management Console (MMC). There is no default console that includes this snap-in; you need to open a blank MMC window or an existing console and manually add the snap-in.

Note that WMI is sensitive to User Account Control (UAC) on versions of Windows that support that security feature. If you plan to query WMI information from a remote computer, you must have a true Administrator security token. Typically, that means you either need to run Windows PowerShell "as Administrator" (ensuring that you obtain the necessary elevated privileges) or you need to provide alternate credentials when establishing a WMI connection.

**Note:** For the purposes of the labs in this course, ensure that you always run Windows PowerShell by right-clicking its icon and selecting Run As Administrator. Doing so may generate a UAC prompt, which helps ensure that you obtain the elevated privileges needed to complete the lab tasks.

## Demonstration: WMI Security

- Learn how to view and configure security on WMI namespaces



### Key Points

In this demonstration, you will learn how to view and configure security on WMI namespaces.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod04\Democode\WMI Security.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Discussion: WMI Explorers

- What sorts of management information might you want to query from WMI?
- Given the type of information, what might a related WMI class be named?
- Can you locate the class in the WMI repository, perhaps using a WMI explorer or browser tool?



### Key Points

As mentioned previously, being able to "explore" the WMI repository, and your computer's instances, is one way that you can locate classes related to a specific task.

What sorts of management information might you want to query from WMI? Given the type of information, what might a related WMI class be named? Can you locate the class in the WMI repository, perhaps using a WMI explorer or browser tool?

## Demonstration: Exploring WMI

- See how to use a WMI explorer tool to browse the WMI repository on a computer



### Key Points

In this demonstration, you will see how to use a WMI explorer tool to browse the WMI repository on a computer.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod04\Democode\Exploring WMI.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Listing Classes from Within the Shell

- The Get-WmiObject cmdlet
  - Can be used to list the classes in the default namespace for either the local computer or a remote computer
  - Can use alternate credentials for remote connections
  - Can generate a list of all namespaces

```
Get-WmiObject -namespace root\cimv2 -list
```

```
Get-WmiObject -namespace root\cimv2 -list  
-computername SEA-DC1
```

```
Get-WmiObject -namespace root\cimv2 -list  
-computername SEA-DC1 -credential  
CONTOSO\Administrator
```

```
Get-WmiObject -namespace root -class  
"__namespace" | ft name
```

### Key Points

Windows PowerShell can also help you explore the classes in a given namespace because it can list all of the classes present in a namespace.



You may want to follow along and run the following commands in one To list the classes in the default namespace, run this command:

```
Get-WmiObject -namespace root\cimv2 -list
```

**Note:** Although Windows PowerShell permits you to use either the forward or backward slash as a path separator for the file system, WMI is somewhat pickier. You should always use the backslash in WMI namespace names.

You can replace the Root\Cimv2 namespace with another namespace name to see the classes in that other namespace. Add the –computerName parameter to list the classes in a namespace on a remote computer:

```
Get-WmiObject -namespace root\cimv2 -list -computername LON-DC1
```

Normally, the Get-WmiObject cmdlet uses the credentials of whatever user account you used to open the Windows PowerShell window. This is the only account that can be used when querying WMI locally; WMI itself does not permit the use of alternate credentials for local connections. For remote connections, however, you can add the –credential parameter to specify an alternate account name. Provide a user name in DOMAIN\USERNAME format:

```
Get-WmiObject -namespace root\cimv2 -list -computername LON-DC1  
-credential CONTOSO\Administrator
```

Windows PowerShell prompts you for the account password. Note that your credentials are not passed in clear text across the network; WMI uses normal Windows authentication mechanisms (usually Kerberos in a domain environment) to process the credential.

**Note:** There is, by design, no easy way to provide a password on the command-line. To prevent you from hardcoding a password in clear text within a script, Windows PowerShell does not accept passwords as part of the –credential parameter. You can, however, create a reusable credential object that can be passed to the –credential parameter, meaning you will be prompted for the password only when you initially create the object. You will learn that technique later in this course.

Finally, to see a list of all namespaces, run the following command in Windows PowerShell:

```
get-wmiobject -namespace root -class "__namespace" | ft name
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 2

# Using Windows Management Instrumentation

- Manipulate WMI objects within the Windows PowerShell pipeline
- Retrieve WMI information using Windows PowerShell
- Specify WMI filtering criteria



Windows PowerShell makes most of WMI available through two straightforward cmdlets, giving you an easy way to retrieve WMI information from local and remote computers as well as a way to invoke WMI methods.

Keep in mind that Windows PowerShell represents WMI information in the form of objects, just as it does for almost everything else done in the shell. That means the cmdlets you have already learned for sorting, filtering, grouping, formatting, exporting, comparing, and so forth can all be used in conjunction with WMI within the shell.

### Lesson Objectives

After completing this lesson, you will be able to:

- Manipulate WMI objects within the Windows PowerShell pipeline.
- Retrieve WMI information using Windows PowerShell.
- Specify WMI filtering criteria.

## Querying WMI

- The Get-WmiObject cmdlet also
  - Retrieves WMI class and instance information from local and remote computers
  - Is aliased with the command gwmi

**Get-WmiObject Win32\_Service  
gwmi Win32\_Service**

- Functionality within WMI classes often overlaps with existing PowerShell cmdlets
  - In some cases, WMI provides more information or exposes additional methods
  - WMI Properties and methods are exposed using Get-Member

**Gwmi Win32\_Service | Get-Member**

### Key Points

The **Get-WmiObject** cmdlet takes care of retrieving WMI information from either the local computer or from one or more remote computers. The **-class** parameter specifies the name of the class you want to retrieve; the default behavior is to retrieve all instances of that class. If the class is not in the Root\Cimv2 namespace, use the **-namespace** parameter to specify the name of the namespace in which the class can be found. Use the **-computername** parameter to specify a remote computer name, and the **-credential** parameter to specify alternate credentials for remote connections. You can query only one class at a time.

The cmdlet retrieves the specified instances and puts them into the pipeline. For example:

```
Get-WmiObject Win32_Service
```

**Note:** A built-in alias for Get-WmiObject is *gwmi*.

Note that some WMI classes provide overlapping functionality with other Windows PowerShell cmdlets—such as Get-Service. However, you may notice differences in how WMI works and the information it presents. Get-Service, for example, uses Microsoft .NET Framework to retrieve information about services; that information does not include details such as the service's startup mode and its logon account because that information was not included in the Framework's implementation. The WMI Win32\_Service class, however, does include the startup mode and logon account information. You can see these properties by piping a class instance to Get-Member:

```
Get-WmiObject Win32_Service | Get-Member
```

You can also see all properties of all class instances by using Format-List:

```
Get-WmiObject Win32_Service | Format-List *
```

## Multiple Computers?

- Get-WmiObject also has an extremely useful ability in interrogating multiple computers in a single command
  - This ability uses its `-computerName` parameter in combination with a comma-separated list

```
Gwmi Win32_Service  
-computerName "LON-DC1", "SEA-DC2"
```

- Multiple computers can also be identified in a text file, which is read into Get-WmiObject using the `Get-Content` cmdlet
- Lists of multiple computers are interrogated sequentially
  - Nonresponsive or powered down computers will cause a delay, present an error message, and continue processing

```
Gwmi Win32_Service  
(Get-Content c:\names.txt)
```

### Key Points

Pay close attention to the `-computerName` parameter in the help file for `Get-WmiObject`. It is listed as follows in the syntax summary:

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

The `<string[]>` value type indicates that the parameter can accept multiple computer names. One way to specify them is to provide a comma-separated list because Windows PowerShell automatically interprets these lists as collections of objects:

```
-computerName "LON-DC1", "SEA-DC2"
```

However, anything that results in a collection of string objects can be given to the `-computerName` parameter. For example, you might create a text file that contains one computer name per line:

```
LON-DC1  
SEA-DC2  
NYC-SRV7
```

You could then use the **Get-Content** cmdlet to read that text file. The cmdlet treats each line of the file as a string object, which is what the `-computerName` parameter needs:

```
-computerName (Get-Content c:\names.txt)
```

**Note:** The parentheses force the shell to execute `Get-Content` first. The results of that cmdlet are then passed to the `-computerName` parameter.

When connecting to multiple remote computers, the Get-WmiObject cmdlet can still accept alternate credentials in the –credential parameter. However, that same credential will be used for all computers specified.

Multiple connections are performed sequentially, not in parallel, so a long list of names might take some time to execute. If the cmdlet is unable to reach a computer—because it is offline, connectivity is blocked, or you do not have permissions—that is considered a nonterminating error, which normally means that the cmdlet displays an error message and continues trying the next computer name in the list.

The objects returned by Get-WmiObject always have a \_\_SERVER property attached to them, which contains the name of the computer that the object came from. This property can be useful when you are querying multiple computers because it enables you to sort and group the output objects by computer name:

```
Get-WmiObject Win32_Service -computer (Get-Content c:\names.txt) |  
    Sort __SERVER | Format-Table -groupBy __SERVER
```

The Get-WmiObject cmdlet always uses Remote Procedure Calls (RPCs) to contact remote computers; it does not use Windows PowerShell's native remoting capability. For this reason, Get-WmiObject can contact any remote computer that has the WMI service running, even if Windows PowerShell is not installed on the remote computer.

Another way to query multiple computers is to use the ForEach-Object cmdlet. Remember that the Get-Content cmdlet reads a text file and returns each line as an object. ForEach-Object would enable you to enumerate through those objects. If each line of the text file was a single computer name, you would be enumerating through computer names. An advantage to this technique is that ForEach-Object can execute multiple commands against each computer:

```
Gc c:\names.txt | ForEach-Object {  
    gwmi win32_computersystem;  
    gwmi win32_operatingsystem } |  
    select buildnumber,servicepackmajorversion,totalphysicalmemory
```

The ForEach-Object script block contains two WMI queries, each returning a different object. Both objects are placed into the pipeline and piped to Select-Object. Select-Object then selects the specified three properties. The output looks something like this:

```
buildnumber servicepackmajorversion totalphysicalmemory  
-----  
3220758528  
7600      0
```

If you look closely, you notice that the output is not combined onto a single line. The first object in the pipeline is a Win32\_ComputerSystem object, which does not have a BuildNumber or a ServicePackMajorversion property—so those columns of the first line are blank. The second object in the pipeline does not have a TotalPhysicalMemory property, and so that column of the second line is blank. Shortly, you will see an example that combines these pieces of information onto a single line of output.

## Demonstration: Querying WMI

- Review how to query WMI from within the shell



### Key Points

In this demonstration, you will review how to query WMI from within the shell.

### Demonstration steps:

1. Start virtual machines **LON-DC1** and **LON-SVR1** then log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod04\Democode\Querying WMI.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Tips and Tricks

- Use PowerShell's computational capabilities to gather and format data to exactly how you need it

- For example, use calculations to display free space in gigabytes rather than bytes:

```
gwmi win32_logicaldisk |  
    select deviceid, drivetype,  
    @{Label='freespace(gb)';Expression={$_.freespace/1GB}}
```

- Or, supply values for one query by using another query:

```
Gwmi win32_operatingsystem |  
    select caption,  
    @{Label='PhysMemory';  
    Expression={(gwmi  
    win32_computersystem).totalphysicalmemory}}
```

## Key Points

You have already learned how to use the `Select-Object` cmdlet to add custom properties to an object. The `Expression` portion of a custom property definition can contain nearly any Windows PowerShell code, including a completely new cmdlet. For example, the following command would retrieve the computer's logical disks and display free space information in gigabytes rather than bytes:

```
gwmi win32_logicaldisk |  
    select deviceid, drivetype,  
    @{Label='freespace(gb)';Expression={$_.freespace/1GB}}
```

A slightly more advanced version of this technique might use the shell's `-as` operator to cast the new free space value as an integer. Because integers do not have a fractional component, the shell rounds the free space to the nearest integer, eliminating the decimal portion of the value:

```
gwmi win32_logicaldisk |  
    select deviceid, drivetype,  
    @{Label='freespace(gb)';Expression={$_.freespace/1GB -as [int]}}
```

Another variation of this technique enables you to combine information from two or more WMI classes onto a single line of output. For example, suppose you want to retrieve a computer's Windows version and the amount of physical memory installed. You might start by querying the `Win32_OperatingSystem` class:

```
Gwmi win32_operatingsystem | select caption
```

You could then add a custom property to the output, and the value for that custom property would be a whole new WMI query to the `Win32_ComputerSystem` class:

```
Gwmi win32_operatingsystem |  
    select caption,
```

```
@{Label='PhysMemory';
Expression={(gwmi win32_computersystem).totalphysicalmemory}}
```

What's happening here? The Expression portion of the custom property is executing a new WMI query. Notice that the query is contained within parentheses. That allows the results of the query to be treated as an object, particularly since this query returns only a single WMI object. (There is, after all, only one computer system on any given computer.) The dot that follows the closing parentheses tells the shell that you want to access a member of the WMI object that the query returned; the member we are accessing is the TotalPhysicalMemory property. That property's value, then, becomes the value for our custom PhysMemory property.



Try running the preceding command to see the results.

What if you were trying to connect to a remote computer? One technique would be to simply specify the computer name for both WMI queries:

```
Gwmi win32_operatingsystem -comp Server1 |
    select caption,
    @{Label='PhysMemory';
    Expression={
        (gwmi win32_computersystem -comp Server1).totalphysicalmemory
    }}
```

However, a better technique would be to specify the computer name only for the original WMI connection. After a WMI object is available, you can use the special `_SERVER` property to access the computer name. That means you would have to change the computer name in only one place to query a different computer:

```
Gwmi win32_operatingsystem -comp Server1 |
    select caption,
    @{Label='PhysMemory';
    Expression={
        (gwmi win32_computersystem -comp $_._SERVER).totalphysicalmemory
    }}
```

In this revised example, the `$_` placeholder—which is valid within the Expression script block—contains the WMI object that was piped into `Select-Object`. That placeholder is followed by a period, which tells WMI that we want to access a member of the object contained in the placeholder. We access the `_SERVER` property, which contains the computer name that the WMI object came from, and use that computer name as the value for the `-computerName` parameter in the second WMI query.

## Filtering the Data

- Gathering large amounts of data is computationally expensive with WMI. Filtering that data is important to creating optimized queries.
  - Filtering with WMI can occur using Where-Object (inefficient) or through the -filter parameter (optimized).
  - Get-WmiObject's -filter parameter submits the filter criteria to the WMI Service, which performs the filtering on behalf of the request.

```
Gwmi Win32_Service | Where { $_.Name -eq 'BITS' }  
Gwmi Win32_Service -filter "Name = 'BITS'"
```

- Using the -filter parameter requires a slightly different syntax than traditional PowerShell comparison operators.
  - Refer to the special filter syntax in your student guide for details and operators.

### Key Points

In some instances, you might not want every single class instance. For example, perhaps you want to query only a single service, BITS, from WMI. One method would be to use the Where-Object cmdlet:

```
Gwmi Win32_Service | Where { $_.Name -eq 'BITS' }
```

However, this technique has a disadvantage in that it must query every class instance, bring them into the shell, and then run through them all to see which ones meet your filtering criteria. When you are retrieving a large number of objects, especially from a remote computer, this can be very time-consuming and processor-intensive.

The Get-WmiObject cmdlet offers a -filter parameter, which submits your filter criteria to the WMI service. The WMI service can filter information much more quickly (it is designed to do so), and it returns only those instances that fit your criteria. Unfortunately, the WMI service does not accept Windows PowerShell-style comparison operators; you have to remember a slightly different syntax for your filter criteria if you use this technique:

```
Gwmi Win32_Service -filter "Name = 'BITS'"
```

Some notes on this filter syntax:

- Strings must be enclosed within single quotation marks. For that reason, the entire filter criteria is usually enclosed in double quotation marks, as shown above.
- Comparison operators are:
  - = (equality)
  - <> (inequality)
  - >= (greater than or equal to)

- <= (less than or equal to)
  - > (greater than)
  - < (less than)
  - LIKE (which permits wildcards)
- Do not use the \$\_ placeholder—that placeholder has meaning only to Windows PowerShell, not to WMI. For WMI filter criteria, list the property name.
  - You can use the keywords AND an OR to specify multiple criteria.

The LIKE operator is especially flexible. When using it:

- Use % as a wildcard: "Name LIKE '%windows%'"
- Use \_ as a single-character wildcard
- Use [ and ] to enclose a range or set of characters: "DriveType LIKE '[abcdef]'"

**Note:** WMI, like Windows PowerShell, is not usually case-sensitive. Keywords such as *LIKE* can also be typed as *like*, and string comparisons such as service names are usually case-insensitive. Some WMI providers may be case-sensitive, but it is considered an unusual implementation.

A best practice is to *filter left*, that is, to specify filtering criteria as far to the left of the Windows PowerShell command line as possible. When you can filter by using a –filter parameter of a cmdlet, doing so is preferable to using the Where-Object cmdlet for the same filtering task.

MCT USE ONLY. STUDENT USE PROHIBITED

## Already Have a Query?

- WMI has its own query language called WMI Query Language or WQL
  - These queries can also be used with Get-WmiObject by using the –query parameter

```
Gwmi -query "SELECT * FROM Win32_Process"
```

- There tends to be no performance advantage or disadvantage in using the –query parameter over other forms of Get-WmiObject

### Key Points

Sometimes, you may have a WMI Query Language (WQL) query from someone else's example, or from an older technology such as VBScript, for which WQL was the primary method of querying WMI data. Windows PowerShell enables you to use these WQL queries. Simply pass the query string to the –query parameter of Get-WmiObject:

```
Gwmi -query "SELECT * FROM Win32_Process"
```

The –query parameter can be used in conjunction with the –namespace, –credential, –computername, and many other parameters, but may not be used in conjunction with the –class parameter because the query string must specify the class being queried.

**Note:** Remember, the help file for Get-WmiObject gives you more information about its parameters.

There is no performance advantage or disadvantage to using the –query parameter over another form of Get-WmiObject.

## System Properties

- Some properties have names that begin with a double-underscore
  - \_\_SERVER
  - \_\_PATH
- These properties are attached by WMI itself, and contain valuable information
  - \_\_SERVER contains the name of the computer where the WMI object came from
  - \_\_PATH contains a designator to uniquely reference a WMI object
- You can use these system properties in your queries to gather and compute information, in the same manner as standard results

### Key Points

If you pipe a WMI object to `Format-List *`, you may notice several properties whose names begin with a double underscore, such as `_SERVER` and `_PATH`:

```
Get-WmiObject Win32_Process | Format-List *
```

These system properties are attached by WMI itself, and they can contain valuable information. For example, you have already seen how the `_SERVER` property contains the name of the computer that the WMI object came from. This is true even if you specified an alias such as `localhost`, or an IP address, for the WMI connection:

```
Gwmi Win32_BIOS -computer localhost | Format-List *
```

`_SERVER` always contains the actual computer name. The `_PATH` property can also be useful: It contains a designator that can be used to uniquely reference a WMI object, and you can even use it to re-retrieve that WMI object if you need to.

## WMI and ForEach-Object

- It is common to retrieve more than one WMI object. It is also common to need to accomplish some action with each of those objects as they are retrieved.
- Unfortunately, WMI is not a part of Windows PowerShell. Thus, the shell contains limited cmdlets for working with WMI objects.
  - In some situations, PowerShell itself may have cmdlets that accomplish the task you need.
  - In other situations, you may be forced to resort to ForEach-Object to accomplish the necessary task.

**Gwmi Win32\_Process | ForEach-Object { Something }**

- It is a best practice to use any available PowerShell cmdlets prior to resorting to ForEach-Object.

### Key Points

It's common to retrieve one or more WMI objects and need to do something with each one individually. Because WMI is not a part of Windows PowerShell, the shell contains limited cmdlets for working with WMI objects. In *most* cases, the shell's core cmdlets can manipulate WMI objects as required to complete your task. In other scenarios, however, you may need to pipe WMI objects to the ForEach-Object cmdlet to perform some specific action with each WMI object in turn:

```
Gwmi Win32_Process | ForEach-Object { $_. }
```

This example outputs each WMI object to the pipeline, which means ForEach-Object isn't really performing a useful task. However, you can put nearly any Windows PowerShell commands that you want within that script block—even multiple commands.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: WMI Filtering and Queries

- Learn how to use WMI filtering and the –query parameter



### Key Points

In this demonstration, you will learn how to use WMI filtering and the –query parameter.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod04\Democode\WMI Filtering and Queries.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

# Lab: Using Windows Management Instrumentation in Windows PowerShell

- Exercise 1: Building Computer Inventory
- Exercise 2: Discovering WMI Classes and Namespaces
- Exercise 3: Generating a Logical Disk Report for All Computers
- Exercise 4: Listing Local Users and Groups

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2	LON-CLI1
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 45 minutes**

## Estimated time: 45 minutes

You work as a systems administrator, and you need to perform certain tasks against the computers, users, and groups that you manage. You need to check inventory of your computers, including the operating system versions, service pack versions, and asset tags. Your organization uses the BIOS serial number as an asset tag tracking system. You need to monitor logical drive space on multiple remote computers. You also need to generate reports showing local users and groups on those machines for audit purposes.

### Lab Setup

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

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Disable the **Windows Firewall** on LON-DC1, LON-SVR1, and LON-SVR2 and remain logged on to the virtual machines.
3. Disable the **Windows Firewall** on LON-CLI1 and also remain logged onto this virtual machine.
4. On LON-CLI1, open the Windows PowerShell console. All PowerShell commands will be run within the Windows PowerShell console.
5. On virtual machine LON-CLI1, create a text file containing the names of all computers in the domain, one on each line, and save it as **C:\users\administrator\documents\computers.txt**. You can either do this in the PowerShell command line if you are able or manually. Its contents should look as follows:
  - LON-DC1.contoso.com

MCT USE ONLY. STUDENT USE PROHIBITED

- LON-SVR1.contoso.com
- LON-SVR2.contoso.com
- LON-CLI1.contoso.com

## Exercise 1: Building Computer Inventory

### Scenario

You work as a system administrator. Periodically you need to inventory your domain computers. The inventory information you require includes the operating system version, the service pack version, and the asset tag. Your organization uses the BIOS serial number as the asset tag for all computer systems.

The main tasks for this exercise are carried out on LON-CLI1 and are as follows:

1. Retrieve the operating system information for the local computer.
2. Extract specific version information from the operating system object.
3. Retrieve operating system version numbers from the local computer "remotely".
4. Retrieve operating system version numbers from multiple computers in one command.
5. Export operating system version numbers (both the operating system version and the service pack version) for multiple computers to a .csv file.
6. Retrieve BIOS serial numbers from multiple computers.
7. Create a custom object for each computer containing all inventory information.
8. Export the results of the last command to a file.

► **Task 1: Retrieve the operating system information**

1. Get the operating system information from the local computer using **Get-WmiObject** with the **Win32\_OperatingSystem** class.
2. Show all properties of the Win32\_OperatingSystem object you just retrieved.

► **Task 2: Extract version information from the operating system object**

1. Show all properties of the Win32\_OperatingSystem object with *version* in their name.
2. Generate a table showing the **\_SERVER**, **Version**, **ServicePackMajorVersion**, and **ServicePackMinorVersion** properties of the **Win32\_OperatingSystem** object.

**Hint:** The **\_SERVER** property starts with two underscore characters.

► **Task 3: Retrieve the local operating system information "remotely"**

1. Repeat the last command, passing the **LON-CLI1** computer name to the **ComputerName** parameter of the **Get-WmiObject** cmdlet.
2. Show the help information for the **ComputerName** parameter of the **Get-WmiObject** cmdlet. Note that **ComputerName** accepts an array of strings.

► **Task 4: Retrieve the operating system information for all computers**

1. Read the list of domain computer names from the **computers.txt** file in your documents folder.
2. Modify the last command you ran to retrieve operating system information so that it shows the information for all computers in the domain.

► **Task 5: Export the operating system information for all computers**

1. Read the help information for the **Select-Object** cmdlet.

2. Replace the call to **Format-Table** in the last **Get-WmiObject** command pipeline with **Select-Object**, maintaining the list of properties that are required.
  3. Export the operating system information to a .csv file called **Inventory.csv**.
- **Task 6: Retrieve the BIOS information for all computers**
- Retrieve the BIOS information for all computers using the **Win32\_BIOS** class.
- **Task 7: Create a custom object for each computer containing all inventory information**
1. Read the help information and examples for the **Select-Object** cmdlet so that you understand what it is used for and how it works.
  2. Combine the operating system version information and BIOS serial numbers into one set of results using **Select-Object**.

**Hint:** This is a complex command. You need to execute Get-WmiObject and use Select-Object to create a custom column. The expression for that custom column is another call to Get-WmiObject. Refer to the course material for an example of this complex technique.

► **Task 8: Export the inventory report**

- Export the results of the last command to a .csv file called **Inventory.csv**.

**Results:** After this exercise, you should have successfully retrieved operating system and BIOS information from all computers in your domain, built custom objects using Select-Object, and generated an inventory report in a .csv file.

## Exercise 2: Discovering WMI Classes and Namespaces

### Scenario

You are retrieving computer information remotely using WMI. You need to be able to discover classes and namespaces on a remote computer so that you can get the data you need.

The main tasks for this exercise are carried out on LON-DC1 and are as follows:

1. View WMI classes in the default namespace on LON-DC1.
2. Find WMI classes using wildcards on LON-DC1.
3. Enumerate the list of top-level WMI namespaces on the LON-DC1 computer.

► **Task 1: View WMI classes in the default namespace**

1. On LON-DC1, read help information for the **List** parameter in the **Get-WmiObject** cmdlet.
2. Get a list of all **WMI** classes in the default namespace on the LON-DC1 computer.

► **Task 2: Find WMI classes using wildcards**

1. Find all **WMI** classes in the default namespace on LON-DC1 that contain *Printer* in their class name.
2. Find all **WMI** classes in all namespaces on LON-DC1 that contain *Printer* in their class name.

► **Task 3: Enumerate top-level WMI namespaces**

1. Enumerate the top-level **WMI** namespaces by retrieving **WMI** objects of class **\_NAMESPACE** in the root namespace on LON-DC1.

**Hint:** The **\_NAMESPACE** class name begins with two underscore characters.

2. Generate a list of only the top-level WMI namespace names by using **Get-WmiObject** with **Select-Object**.

**Hint:** Sometimes, a property is a collection of other objects. **Select-Object** can expand that property into a list of those objects by using the **-ExpandProperty** parameter. For example, **-ExpandProperty Name** expands the **Name** property.

**Results:** After this exercise, you should be able to find WMI classes in a specific namespace, find WMI classes using a wildcard search, and find WMI namespaces on local or remote computers.

## Exercise 3: Generating a Logical Disk Report for All Computers

### Scenario

To monitor disk-space usage for computers in your organization, you want to use PowerShell to retrieve and process WMI information.

The main tasks for this exercise are carried out on LON-CLI1 and are as follows:

1. Learn how to discover the WMI class used to retrieve logical disk information.
2. Retrieve logical disk information from the local machine.
3. Apply a server-side filter to filter the logical disks to include only hard disks and gather disk information for all computers in your domain.
4. Add a calculated value to your report showing percent-free information for hard disks for all computers in your domain.
5. Learn how to discover information related to WMI classes.

#### ► Task 1: Discover the WMI class used to retrieve logical disk information

- Find all **WMI** classes in the default namespace on the local machine that contain *LogicalDisk* in the class name.

**Hint:** If you examine the help for `Get-WmiObject`, you see that the `-class` parameter is positional and is in position 2. That means you do not have to type the `-class` parameter name; you can simply provide a value, if you place it in the correct position.

#### ► Task 2: Retrieve logical disk information from the local machine

- Get all logical disk drives for the local machine using the **Get-WmiObject** cmdlet.

#### ► Task 3: Retrieve logical disk information for hard disks in all computers in your domain

1. Read help information for the **Filter** parameter of the **Get-WmiObject** cmdlet.
2. Read the list of computer names in your domain from the **computers.txt** file in your documents folder.
3. Get all hard disks for all computers in your domain using the **Get-WmiObject** cmdlet. The **DriveType** property value for hard disks is **3**. Show the results in a table with the **computer name**, **device id**, and **free-space** information.

#### ► Task 4: Add a calculated PercentFree value to your report

- Add a calculated parameter to your hard disk table that has the following properties:
  - Label: 'PercentFree'
  - Expression: `{$_._FreeSpace / $_._Size}`
  - FormatString: '{0:#0.00%}'

**Hint:** You have already seen examples of the structure necessary to create a custom or calculated column. In this task, you are doing the same thing. However, in addition to the **Label** and **Expression** elements that you have seen before, you are adding a **FormatString** element.

► Task 5: Discover WMI information related to the logical drives

1. Show only the first hard drive found in your report.
2. Invoke the **GetRelated()** method on the first drive, and show the results in a table containing **\_CLASS** and **\_RELPATH**.

**Hint:** **\_CLASS** and **\_RELPATH** both begin with two underscore characters. These are properties of the WMI objects, just like any other properties. The underscore characters indicate that they are system properties, added by WMI and contain information that helps WMI locate and manage the objects.

MCT USE ONLY. STUDENT USE PROHIBITED

**Hint:** Remember that Get-WmiObject returns objects and that some objects have *methods* that perform actions. GetRelated() is one such method. For example, you could enclose an entire WMI command in parentheses to execute the GetRelated() method of the object(s) returned by that command:

```
(Get-WmiObject your parameters go here | Select -First 1).GetRelated()
```

That Select-Object command ensures that only the first returned object is used.

**Results:** After this exercise, you should know how to check logical disk free-space information using WMI, how to add calculated properties to a report, and how to find related WMI information from your WMI data.

## Exercise 4: Listing Local Users and Groups

### Scenario

Corporate regulations require that you maintain an inventory of local users and groups on computers in your domain for audit purposes.

The main tasks for this exercise are carried out on LON-CLI1 and are as follows:

1. Find the WMI classes used to retrieve local users and groups.
2. Generate a report showing all local user accounts and groups from all computers in your domain. This report should contain the WMI object class to identify the object type as well as the computer name, user or group name, and SID.

► **Task 1: Find the WMI classes used to retrieve local users and groups**

1. Search for **WMI** classes representing local users in the default namespace using **Get-WmiObject**.
2. Search for **WMI** classes representing local groups in the default namespace using **Get-WmiObject**.

► **Task 2: Generate a local users and groups report**

1. Retrieve a list of all local user accounts on the local computer and identify the properties containing the WMI object class name, the computer name, the user name, and the SID.
2. Retrieve a list of all local groups on the local computer and identify the properties containing the WMI object class name, the computer name, the user name, and the SID.
3. Read the basic help information for the **ForEach-Object** cmdlet. Note that **ForEach-Object** allows you to process a collection of objects one at a time.
4. Generate a single table showing all local users and groups from all computers in the computers.txt file that is in your documents folder. Format the table output so that it contains only the object type, the computer name, the user or group name, and the SID.

**Hint:** If you want to include two (or more) independent shell commands into a single {script block}, separate the commands by using a semicolon.

**Results:** After this exercise, you should be able to retrieve local user and group account information from local and remote computers and generate a report combining this information into a single table.

## Lab Review

1. How do you list WMI classes in a specific namespace?
2. What is the WMI class to retrieve operating system information called?
3. How do you use a server-side filter when using Get-WmiObject?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- Where is the best place to use filtering criteria with WMI?
- When can alternate credentials be used with WMI connections?
- What protocol does WMI use, and how can you enable it to pass through a local firewall?



### Review Questions

1. Where is the best place to use filtering criteria with WMI?
2. When can alternate credentials be used with WMI connections?
3. What protocol does WMI use, and how can you enable it to pass through a local firewall?

### Common Issues Related to WMI

Identify the causes for the following common issues related to the core cmdlets and fill in the troubleshooting tips. For answers, refer to relevant lessons in the module.

Issue	Troubleshooting tip
You cannot connect to a remote computer by using WMI.	
A WMI class instance does not have the same properties as described on the class' documentation page.	

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 5

## Automating Active Directory® Administration

### Contents:

Lesson 1: Active Directory Automation Overview	5-3
Lesson 2: Managing Users and Groups	5-10
Lab A: Managing Users and Groups	5-14
Lesson 3: Managing Computers and Other Directory Objects	5-19
Lab B: Managing Computers and Other Directory Objects	5-22

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Overview

- Use the AD: drive in Windows PowerShell
- Identify Active Directory cmdlets in Windows PowerShell
- Perform key Active Directory management tasks related to users, groups, computers, and organizational units, by means of Windows PowerShell cmdlets



Active Directory® is the technology where many administrators spend the majority of their time, completing day-to-day administrative tasks such as adding users and updating directory objects. With the introduction of Active Directory-focused cmdlets in Windows Server® 2008 R2, those administrators have the potential to save a great deal of time and energy by using Windows PowerShell® to automate many of their more time-consuming or repetitive tasks. Automation can also help improve security and consistency because it is less prone to repeated human error than manual administration.

### Module Objectives

After completing this module, you will be able to:

- Use the AD: drive in Windows PowerShell.
- Identify Active Directory cmdlets in Windows PowerShell.
- Perform key Active Directory management tasks related to users, groups, computers, and organizational units, by means of Windows PowerShell cmdlets.

## Lesson 1

# Active Directory Automation Overview

- Identify prerequisites for using the Microsoft Active Directory cmdlets in a domain
- Explain the purpose and use of the AD: drive
- List the cmdlets included in the ActiveDirectory module



This lesson will help you understand the approach used by the Active Directory cmdlets. This lesson introduces the first major set of add-in commands in this course. It will help you develop the skills that you need to discover, explore, learn, and use other add-in commands, whether they come with Windows Server 2008 R2 or with another Microsoft or third-party software product.

A key goal of this course is to teach you how to be more self-reliant when using Windows PowerShell, even if you are working with commands that were not addressed in this course. This lesson is the first major step toward accomplishing that goal.

### Lesson Objectives

After completing this lesson, you will be able to:

- Identify prerequisites for using the Microsoft Active Directory cmdlets in a domain.
- Explain the purpose and use of the AD: drive.
- List the cmdlets included in the Active Directory module.

MCT USE ONLY. STUDENT USE PROHIBITED

## Active Directory Administration

- Nearly all Active Directory administration in PowerShell is accomplished using the ActiveDirectory module.
  - This module is installed on all Domain Controllers. It is also included as part of the Remote Server Administration Tools (RSAT) for Windows 7.
- The ActiveDirectory module includes cmdlets that facilitate virtually every activity in Active Directory administration.
  - Its cmdlets provide the functionality that powers the graphical Active Directory Administrative Center console.
  - Its cmdlets communicate with a web service that is a part of Active Directory in Windows Server® 2008 R2.
  - This same web service can be added to Windows Server 2003 and Windows Server 2008 by downloading and installing the Active Directory Management Gateway Service.
- The web service needs to be installed to only a single Domain Controller in your local site.

### Key Points

Windows Server 2008 R2 introduces an Active Directory module for Windows PowerShell. This module is installed on all domain controllers, and is included as part of the Remote Server Administration Toolkit (RSAT) for Windows® 7. The module includes cmdlets that facilitate management of Active Directory from the command-line. These cmdlets also provide the functionality that powers the new graphical Active Directory Administrative Center console.

These new cmdlets communicate with a Web service that is part of Active Directory in Windows Server 2008 R2. The same Web service can be added to Windows Server 2008 and Windows Server 2003 domain controllers by downloading the Web service from Microsoft's Web site.



You will find the Active Directory Management Gateway Service at <http://go.microsoft.com/fwlink/?LinkId=193581>.

You do not need this Web service installed on every domain controller. Having it on one domain controller in the same site as your administrators is sufficient. Also, the cmdlets themselves do not need to be installed on your domain controllers. The cmdlets need to be installed only on the client computer where you will actually run the cmdlets.

## Adding a Module

- The Import-Module cmdlet
  - Can be used to load any external module into PowerShell.
  - Uses the following syntax to add the ActiveDirectory module:

**Import-Module ActiveDirectory**

    - Using this cmdlet imports the module into only the currently-running session. You will need to import it in each session.
    - After it is loaded, the module adds a set of commands for administering Active Directory. You can retrieve the list of commands using:

**Get-Command –module ActiveDirectory**
- The Remove-Module cmdlet will unload the module from the current session.

### Key Points

Use Import-Module to load the Active Directory module into the shell.

```
Import-Module ActiveDirectory
```

Use Remove-Module to remove the module when you are finished with it. The module will remain loaded until you remove it, or until you close your shell session.

You can retrieve a list of the cmdlets included in the module by running:

```
Get-Command –module activedirectory
```



Try loading the module now on LON-DC1, and reviewing the available cmdlets. The list of cmdlets will give you an idea of the tasks you can accomplish.

## The AD: Drive

- Adding the ActiveDirectory module also adds a PSDrive provider.
  - This provider maps the AD: drive to your logon domain.
  - The main purpose of this drive is to provide a security context for executing cmdlets.
- When you run an Active Directory cmdlet, it will automatically use the credentials and domain of the current AD: drive.
  - This eliminates the need to supply credentials for each command.
  - You can map other drives to other domains and credentials. Cmdlets will run using the credentials associated with the current drive.
  - To use a different domain or set of credentials, change to the correct mapped drive, and then begin running cmdlets.

### Key Points

The Active Directory module includes a PSDrive provider. By default, importing the module maps a drive named AD: to your logon domain. The main purpose of the drive is to provide a security context for executing cmdlets.

Each of the Active Directory cmdlets enables you to connect to a remote domain and provide credentials for doing so. The help for each of the Active Directory cmdlets lists the appropriate parameters. For example:

```
[-AuthType {Negotiate | Basic}] [-Credential <PSCredential>]
```

Specifying credentials for every cmdlet could become tedious, which is where the PSDrive provider helps. Try running `help New-PSDrive` when the shell is focused on your C: drive, and again after changing to the AD: drive:

```
Cd c:  
Help New-PSDrive  
Cd AD:  
Help New-PSDrive
```

The cmdlets help changes, providing different parameters when you are in the AD: drive. This alternate version of `New-PSDrive` is specifically designed to map drives to other domains, enabling you to provide alternate credentials. *These credentials are remembered* for as long as the PSDrive mapping remains.

When you run an Active Directory cmdlet, the PSDrive automatically uses the credentials and domain of the current drive. Thus, rather than providing credentials each time you run a cmdlet, you can simply map a drive to the desired domain, change to that drive, and then run cmdlets from there. The cmdlets will automatically use the credentials associated with that drive. To use a different set of credentials, map another drive to the desired domain, change to that drive, and begin running cmdlets.

Only the Active Directory cmdlets work in this fashion. Other cmdlets that support a –credential parameter will not inherit credentials from a PSDrive that is mapped to a domain unless those cmdlets have been specifically designed to do so by their developer.

**Note:** If you frequently work with a given set of domains, you might want to import the Active Directory module and map drives to those domains each time the shell starts. You can do so in a profile script.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: The AD: Drive

- Learn how to import the ActiveDirectory module and use the AD: drive



### Key Points

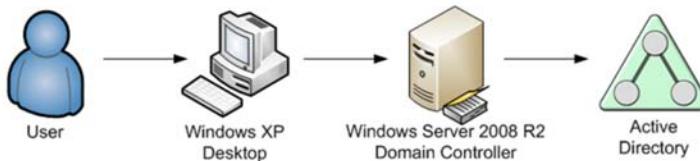
In this demonstration, you will learn how to import the Active Directory module and use the AD drive.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod05\Democode\The AD Drive.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Tip for Earlier Versions of Windows

- Be Aware: The Active Directory cmdlets are designed to be used on Windows Server 2008 R2 and Windows 7 only
  - This means that other operating systems cannot **directly** install and use the cmdlets
  - However, these older operating systems can **indirectly** use the cmdlets of another host
- The process to use another host's cmdlets is called **implicit remoting**, and will be covered later in this course



### Key Points

Keep in mind that the Active Directory cmdlets are designed to be installed on Windows Server 2008 R2 and Windows 7 only. You can, however, access the cmdlets from earlier versions of Windows provided that you have at least one installation of the cmdlets available on your network.

For example, suppose you are using a computer running Windows XP that has Windows PowerShell v2 installed. You might install a Windows Server 2008 R2 domain controller into your domain and enable Windows PowerShell remoting. Doing so would give you an installation of the Active Directory cmdlets as well as a domain controller that the cmdlets can communicate with.

You could then use *implicit remoting* from your Windows XP client computer to access the cmdlets installed on the Windows Server 2008 R2 computer. The cmdlets would behave as if they were installed locally on Windows XP, but they would execute on the remote computer.

This technique will be covered later in this course, in the module on Windows PowerShell remoting.

## Lesson 2

# Managing Users and Groups

- Use Windows PowerShell cmdlets to retrieve, create, enable, disable, modify, move, and remove Active Directory users and groups
- Use Windows PowerShell cmdlets to reset Active Directory user account passwords



User and group management is obviously one of the major administrative tasks in Active Directory. In this lesson, you will learn how to perform user and group management tasks by using cmdlets.

Once again, the skills and techniques you learn here can be applied to other cmdlets that ship with other Microsoft and third-party products.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use Windows PowerShell cmdlets to retrieve, create, enable, disable, modify, move, and remove Active Directory users and groups.
- Use Windows PowerShell cmdlets to reset Active Directory user account passwords.

## Discussion: User and Group Cmdlets

- Which cmdlets are available to manage users and groups?
- Which parameters accept pipeline input when you are creating a new user? Adding a member to a group?



### Key Points

Using the Active Directory cmdlets is essentially a matter of figuring out which cmdlets exist and figuring out how to use each cmdlet. You can use the Get-Command and Help commands to learn about the available cmdlets and how to use them.

We will deliberately spend very little time reviewing the exact syntax for each of the Active Directory cmdlets. A key skill in Windows PowerShell is being able to discover cmdlet names and read their help files on your own. You will develop that skill in this and future modules.

Get into the habit of reading the full help for a cmdlet. Pay attention to which parameters support pipeline binding by using `ByValue` and `ByPropertyName`. Knowing this information can help you develop shortcut commands rather than longer commands or scripts.

As a class, discuss and answer the following questions:

- Which cmdlets are available to manage users and groups?
- Which parameters accept pipeline input when you are creating a new user? Adding a member to a group?

## Filtering

- It is generally a bad idea to query every object in Active Directory at once
  - Doing so is computationally expensive
  - Doing so can impact your Domain Controllers' performance
- Most Active Directory cmdlets have defined a mandatory parameter called `-filter`
  - This `-filter` parameter limits the number of records that the cmdlet will work with
  - It can accept wildcards and PowerShell-style criteria:

```
Get-ADUser -Filter 'Name -like "*SvcAccount"'
Get-ADUser -Filter {Name -eq "GlenJohn"}
```

### Key Points

Generally speaking, it's a bad idea to query every object from the directory at once. Doing so can impose an unacceptable workload on your client computer, your network, and most especially on your domain controllers.

For that reason, most of the Active Directory cmdlets that retrieve objects (those that use `Get` as the verb component of the cmdlet name) have defined a mandatory `-filter` parameter. You can specify `*` for this parameter, but you should generally specify more precise criteria so that you are querying only those objects that you absolutely need.

The `-filter` parameter of the Active Directory cmdlets accepts Windows PowerShell-style criteria:

```
Get-ADUser -Filter 'Name -like "*SvcAccount"'
Get-ADUser -Filter {Name -eq "GlenJohn"}
```

You can read the help file for a cmdlet to see other examples. Most cmdlets can also restrict their operation to a specific subset of the directory, called a *search base*. Read the help file to learn how to specify an organizational unit or other container as a search base.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Managing Users and Groups

- Review how to manage users and groups from within the shell



### Key Points

In this demonstration, you will review how to manage users and group from within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod05\Democode\Managing Users and Groups.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab A: Managing Users and Groups

- Exercise 1: Retrieving a Filtered List of Users from Active Directory
- Exercise 2: Resetting User Passwords and Address Information
- Exercise 3: Disabling Users That Belong to a Specific Group

Logon information

Virtual machine	LON-DC1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are an Active Directory administrator and want to manage your users and groups via PowerShell. You recently upgraded your domain controller to Windows Server 2008 R2 and want to try the new PowerShell Active Directory cmdlets that came with it. In order to handle internal tasks more quickly and be prepared to automate them, you want to learn how to find information in Active Directory as well as how to accomplish basic tasks such as resetting users' passwords, disabling users, and moving objects in Active Directory.

### Lab Setup

Before you begin the lab you must:

1. Start the **LON-DC1** virtual machine and log on with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open the Windows PowerShell console. All commands will be run within the Windows PowerShell console.
3. In the Windows PowerShell console, execute the following command:
  - **Set-ExecutionPolicy RemoteSigned**

## Exercise 1: Retrieving a Filtered List of Users from Active Directory

### Scenario

You want to manage your users and groups via PowerShell. To begin, you want to use Windows PowerShell to retrieve a filtered list of users from Active Directory.

The main tasks for this exercise are as follows:

1. List all modules installed on the local system.
2. Import the Active Directory module and populate the Active Directory Environment.
3. List all commands in the Active Directory module.
4. Retrieve all users matching a specific city and title by using server-side filtering.

► **Task 1: List all modules installed on the local system**

1. Read the help documentation and examples for the **Get-Module** command to familiarize yourself with how it works.
2. Show a list of all modules installed on the LON-DC1 computer by using **Get-Module** with the **ListAvailable** parameter.

► **Task 2: Import the Active Directory module and populate the Active Directory environment**

1. Read the help documentation and examples for the **Import-Module** command so that you understand how it works.
2. Use the **Import-Module** command to import the **Active Directory** module into your current PowerShell session.
3. In the Windows PowerShell console execute the following command:  
**E:\Mod05\Labfiles\Lab\_05\_setup.ps1**.

► **Task 3: List all commands in the Active Directory module**

- Use **Get-Command** to retrieve a list of all commands that were loaded when you imported the Active Directory module.

► **Task 4: Retrieve all users matching a specific city and title by using server-side filtering**

1. Read the help documentation and examples for the **Get-ADUser** command to learn how you can use it to retrieve Active Directory users. Pay extra attention to the **-filter** parameter.
2. Invoke **Get-ADUser** with no parameters to see which parameters are required and learn how to use them.
3. Retrieve a list of all Active Directory users whose office attribute has a value of "Bellevue."
4. Retrieve a list of all Active Directory users whose office attribute has a value of "Bellevue" and whose title attribute has a value of "PowerShell Scripter."

**Results:** After this exercise you should have successfully imported the Active Directory module into PowerShell and used it to retrieve a filtered list of users from Active Directory.

## Exercise 2: Resetting User Passwords and Address Information

### Scenario

You are working in Bellevue, Washington, and you are automating some Active Directory tasks using PowerShell. You need to reset user passwords and change address information for some remote users.

The main tasks for this exercise are as follows:

1. Retrieve a list of remote users.
2. Reset remote user passwords to a specific password.
3. Change remote user address information to your local Bellevue, Washington office.

#### ► Task 1: Retrieve a list of remote users

- Use **Get-ADUser** with the **-filter** parameter to retrieve a list of users whose office attribute is not set to "Bellevue."

#### ► Task 2: Reset remote user passwords to a specific password

1. Review the documentation and examples for the **Read-Host** and **Set-ADAccountPassword** commands. Pay close attention to the **AsSecureString** parameter for **Read-Host** and the **Reset** and **NewPassword** parameters for **Set-ADAccountPassword**.
2. Pass the list of users whose office is not "Bellevue" to the **Set-ADAccountPassword** command and use it with **Read-Host** to set their passwords to a password of **Pa\$\$w0rd** that you are prompted to enter in PowerShell.

#### ► Task 3: Change remote user address information to your local Bellevue, Washington office

1. Look up the help documentation for the **Properties** parameter of **Get-ADUser**.
2. Retrieve a list of users whose office is not "Bellevue." When retrieving the users, use the **Properties** parameter to retrieve the **Office**, **StreetAddress**, **City**, **State**, **Country/Region**, and **PostalCode** attributes. Show the results in a table containing **SamAccountName** along with the other attributes you specified in the **Properties** parameter.
3. Read the help documentation and examples for the **Set-ADUser** cmdlet.
4. Pass the results of the **Get-ADUser** command you just used to the **Set-ADUser** command, and set the following values:

Office: **Bellevue**

StreetAddress: **2345 Main St.**

City: **Bellevue**

State: **WA**

Country: **US**

PostalCode: **95102**

**Results:** After this exercise, you should be able to reset passwords and modify attributes for a filtered list of Active Directory users.

## Exercise 3: Disabling Users That Belong to a Specific Group

### Scenario

Your organization has recently terminated a project at work called "CleanUp" and you need to disable all users that belong to the Active Directory group corresponding to the project.

The main tasks for this exercise are as follows:

1. Retrieve a list of all Active Directory groups.
2. Retrieve a specific Active Directory group named "CleanUp."
3. Retrieve a list of members in the Active Directory group named "CleanUp."
4. Disable the members of the Active Directory group named "CleanUp."

► **Task 1: Retrieve a list of all Active Directory groups**

1. Read the documentation and examples for the **Get-ADGroup** cmdlet.
2. Retrieve a list of all Active Directory groups by using the **Get-ADGroup** cmdlet.

► **Task 2: Retrieve a specific Active Directory group named "CleanUp"**

- Retrieve the Active Directory group named "CleanUp" by using the **Get-ADGroup** cmdlet.

► **Task 3: Retrieve a list of members in the Active Directory group named "CleanUp"**

1. Read the help documentation and examples for the **Get-ADGroupMember** cmdlet.
2. Use the **Get-ADGroup** and **Get-ADGroupMember** cmdlets to retrieve a list of the members in the Active Directory group named "CleanUp."

► **Task 4: Disable the members of the Active Directory group named "CleanUp"**

1. Read the help documentation and examples for the **Disable-ADAccount** cmdlet.
2. Use the **Disable-ADAccount** cmdlet with the **WhatIf** parameter and the list of members of the Active Directory group named "CleanUp" to see what would happen if you were to disable the members in that group.
3. Repeat the last command without the **WhatIf** parameter to actually disable the group members.

**Results:** After this exercise, you should know how to retrieve groups from Active Directory, view their membership, and disable user accounts.

## Lab Review

1. Which common Active Directory cmdlet parameter is used to limit search results to matches based on attributes?
2. Which common Active Directory cmdlet parameter is used to specify the attributes that you want in your query results?
3. How do you add Active Directory functionality to your PowerShell session?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 3

# Managing Computers and Other Directory Objects

- Use Windows PowerShell cmdlets to retrieve and modify Active Directory computer accounts
- Use Windows PowerShell cmdlets to retrieve and view Active Directory fine-grained password policies
- Use Windows PowerShell cmdlets to retrieve computer account information, including operating system version, service pack version, and last logon timestamp



If you are comfortable locating cmdlet names and reading their help files, there is very little that you can't accomplish in the shell. In this lesson, you will focus on managing computers and other directory objects.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use Windows PowerShell cmdlets to retrieve and modify Active Directory computer accounts.
- Use Windows PowerShell cmdlets to retrieve and view Active Directory fine-grained password policies.
- Use Windows PowerShell cmdlets to retrieve computer account information, including operating system version, service pack version, and last logon timestamp.

MCT USE ONLY. STUDENT USE PROHIBITED

## Computer and Other Objects

- The ActiveDirectory cmdlet can also interact with objects other than users, such as:
  - Computer objects
  - Groups
  - Fine-grained password policies
- The cmdlets Get-ADComputer, New-ADFineGrainedPasswordPolicy, and many others interact with these objects in ways that are similar to working with users
  - Remember to pipe objects to Get-Member or Format-List \* to see which objects are available

**Spend time with the help for the ActiveDirectory module's cmdlets to see which administrative actions are exposed**

### Key Points

Active Directory stores a lot more information than you might realize about domain computers.  
Remember to pipe objects to Get-Member or Format-List \* to see what's there.

Also keep in mind that to avoid degrading performance, the Active Directory cmdlets do not usually, by default, retrieve every property that the directory contains. You can always specify that additional attributes or all attributes be retrieved. Read the help for a cmdlet to discover how to do this.

The cmdlets covered in this lesson behave very similarly to those covered in previous lessons.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Computer and Other Objects

- Learn how to manage computer and other directory objects from within the shell



### Key Points

In this demonstration, you will learn how to manage computer objects and other directory objects from within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod05\Democode\Computer and Other Objects.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Managing Computers and Other Directory Objects

- Exercise 1: Listing All Computers That Appear to Be Running a Specific Operating System According to Active Directory Information
- Exercise 2: Creating a Report Showing All Windows Server 2008 R2 Servers
- Exercise 3: Discovering Any Organizational Units That Aren't Protected Against Accidental Deletion

### Logon information

Virtual machine	LON-DC1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 20 minutes**

### Estimated time: 20 minutes

As an Active Directory administrator, in addition to managing users and groups, you need to monitor the servers in your organization. Active Directory contains details identifying servers, and you want to use those details to discover servers and generate reports. To meet new security policies, your company has decided to put more stringent password policies in place. You need to create fine-grained password policies for your organization and heard that PowerShell was the only way to do so. Also, as a senior IT administrator responsible for a team, you want to make sure that your team members don't accidentally delete important information in Active Directory. You want to use a new feature for organizational units OUs that prevents them from accidental deletion.

### Lab Setup

Before you begin the lab you must:

1. Start the **LON-DC1** virtual machine and log on with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open **Windows PowerShell**. All PowerShell commands will be run within the Windows PowerShell console.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 1: Listing All Computers That Appear to Be Running a Specific Operating System According to Active Directory Information

### Scenario

As an Active Directory administrator, in addition to managing users and groups you also need to monitor the servers in your organization. Active Directory contains details identifying servers, and you want to use those details to discover servers.

The main tasks for this exercise are as follows:

1. Import the Active Directory module.
2. List all of the properties for one AD computer object.
3. Find any properties containing operating system information on an AD computer.
4. Retrieve all computers running a specific operating system.

► **Task 1: Import the Active Directory module**

- Import the Active Directory module into your current PowerShell session.

► **Task 2: List all of the properties for one AD computer object**

1. Read the help documentation and examples for the **Get-ADComputer** cmdlet.
2. Show all properties for one computer from Active Directory using the **Get-ADComputer** cmdlet with the **-filter**, **Properties**, and **ResultSetSize** parameters.

► **Task 3: Find any properties containing operating system information on an AD computer**

- Get one computer from Active Directory and show all properties containing operating system information.

► **Task 4: Retrieve all computers running a specific operating system**

1. Get a list of all computers running "Windows Server 2008 R2" using **Get-ADComputer** with the **-filter** parameter.
2. Repeat the last command but include the **OperatingSystem** property in the objects that are returned.

**Results:** After this exercise you should be able to retrieve AD computers that match specific criteria and indicate which properties you want to retrieve for those computers.

## Exercise 2: Creating a Report Showing All Windows Server 2008 R2 Servers

### Scenario

Now that you can discover servers in your organization using Active Directory, your manager would like you to generate a report showing all Windows Server 2008 R2 servers in your organization.

The main tasks for this exercise are as follows:

1. Retrieve all computers running Windows Server 2008 R2.
2. Generate an HTML report showing only the computer name, SID, and operating system details.
3. Generate a CSV file showing only the computer name, SID, and operating system details.

► **Task 1: Retrieve all computers running Windows Server 2008 R2**

- Retrieve all AD computers running the Windows Server 2008 R2 operating system. Include the **OperatingSystem**, **OperatingSystemHotfix**, **OperatingSystemServicePack**, and **OperatingSystemVersion** properties in the computer objects that you retrieve.

► **Task 2: Generate an HTML report showing only the computer name, SID, and operating system details**

1. Read the help documentation and examples for the **ConvertTo-HTML** and **Out-File** cmdlets so that you understand how they work.
2. Rerun the command from Task 1 to retrieve Windows Server 2008 R2 servers. Generate an HTML table fragment by passing the results to the **ConvertTo-HTML** cmdlet. Don't forget to use the **Fragment** parameter.
3. Rerun the command from Task 1 to retrieve Windows Server 2008 R2 servers. Generate an HTML file by passing the results to the **ConvertTo-HTML** and **Out-File** cmdlets. Name the resulting file **C:\OSList.htm**.
4. Use PowerShell to open the **C:\OSList.htm** file in Internet Explorer.

► **Task 3: Generate a CSV file showing only the computer name, SID, and operating system details**

1. Read the help documentation and examples for the **Export-Csv** cmdlet so that you understand how it works.
2. Rerun the command from Task 1 to retrieve Windows Server 2008 R2 servers. Use **Select-Object** to select only the **Name**, **SID**, and **OperatingSystem\*** attributes.
3. Pass the results of the last command to **Export-Csv** to create a CSV file containing the results. Name the resulting file **C:\OSList.csv**.
4. Use PowerShell to open the **C:\OSList.csv** file in Notepad.

**Results:** After this exercise, you should be able to generate HTML and CSV documents containing information you retrieved using PowerShell commands.

## Exercise 3: Discovering Any Organizational Units That Aren't Protected Against Accidental Deletion

### Scenario

As a senior IT administrator responsible for a team, you want to make sure that your team members don't accidentally delete important information in Active Directory. You know about the new feature for OUs that prevents them from accidental deletion and you want to monitor the OUs in your environment to ensure that they are appropriately protected.

The main tasks for this exercise are as follows:

1. Retrieve all organizational units.
2. Retrieve organizational units that are not protected against accidental deletion.

► **Task 1: Retrieve all organizational units**

1. Read the help documentation and examples for the **Get-ADOrganizationalUnit** cmdlet to learn how it works.
2. Get a list of all organizational units in AD. Include the **ProtectedFromAccidentalDeletion** property in the results.

► **Task 2: Retrieve organizational units that are not protected against accidental deletion**

- Use the **Get-ADOrganizationalUnit** cmdlet to get a list of all organizational units whose **ProtectedFromAccidentalDeletion** property is set to **false**.

**Results:** After this exercise, you should be able to use the Active Directory cmdlets to retrieve organizational units from Active Directory.

## Lab Review

1. How can you see a list of all attributes that are available for an Active Directory object?
2. Which parameter can be used to limit the total number of objects returned in an Active Directory query?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- On which operating systems are the Active Directory cmdlets available?
- Which module contains the Active Directory cmdlets?
- What is the purpose of an Active Directory PSDrive?
- Which drive must be active in order to use New-PSDrive to map a new drive to Active Directory?

### Class Discussion

- Common issues related to Active Directory



### Review Questions

1. On which operating systems are the Active Directory cmdlets available?
2. Which module contains the Active Directory cmdlets?
3. What is the purpose of an Active Directory PSDrive?
4. Which drive must be active in order to use New-PSDrive to map a new drive to Active Directory?

### Common Issues Related to Active Directory

Identify the causes for the following common issues related to the core cmdlets, and fill in the troubleshooting tips. For answers, refer to relevant lessons in the module.

Issue	Troubleshooting tip
The Active Directory cmdlet name that you typed cannot be found	
You are prompted to enter "Filter" when running an Active Directory cmdlet	

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 6

## Windows PowerShell® Scripts

### Contents:

<b>Lesson 1:</b> Script Security	6-3
<b>Lesson 2:</b> Basic Scripts	6-15
<b>Lesson 3:</b> Parameterized Scripts	6-22
<b>Lab:</b> Writing Windows PowerShell Scripts	6-30

## Module Overview

- Explain and configure Windows PowerShell script-related security features
- Create and execute basic scripts
- Convert command sequences to parameterized scripts



Although Windows PowerShell® is an excellent command-line shell, repeatedly typing long commands can be time-consuming and error-prone. Windows PowerShell scripts provide you with a way to run a sequence of commands many times with less typing.

Scripts can also provide an easier way to automate lengthy sequences of commands or to automate tasks that require logical decisions, repetitive work, and so forth.

Windows PowerShell script execution is controlled by a set of features that, by default, are configured in a secure state. As you begin to work with scripts, you may need to reconfigure some of these features, taking into account your script execution needs as well as the security policies of your environment.

### Module Objectives

After completing this module, you will be able to:

- Explain and configure Windows PowerShell script-related security features.
- Create and execute basic scripts.
- Convert command sequences to parameterized scripts.

## Lesson 1

# Script Security

- Explain the script security features, including filename extension association, execution policy, and current path searching
- Explain the role of trusted root Certification Authorities (either commercial or private) in shell script security
- Set the shell execution policy locally and in a domain environment
- Explain how to digitally sign a script



The script security features for Windows PowerShell are specifically designed to help prevent some of the scripting-related security problems of older technologies, including Microsoft® Visual Basic® Scripting Edition (VBScript). Windows PowerShell defaults to a secure state and enables you to modify that state to accommodate your scripting needs and a variety of security goals.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the script security features, including filename extension association, execution policy, and current path searching.
- Explain the role of trusted root Certification Authorities (either commercial or private) in shell script security.
- Set the shell execution policy locally and in a domain environment.
- Explain how to digitally sign a script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Script Concerns

- Scripts are good things. Scripts are bad things.
  - Scripts enable the consistent and repeatable administration of IT resources.
  - Scripts can also be an easy way to introduce malware into an environment.
  - Users can be convinced to execute scripts without really understanding what the script is doing or that they are even running a script.
- Windows PowerShell intends to create a “secure by default” environment for running scripts.
  - The shell makes it difficult by default for users to run scripts without realizing they are doing so.
  - Its default configuration may not necessarily be convenient; however, you can reconfigure to make it more convenient.
  - Multiple security options help you find a balance between convenience and security.

### Key Points

The IT industry has had ample experience with the security problems that scripting languages can create. The main security problem is that scripts can be an easy way to introduce malware into an environment, primarily because users can be convinced to execute scripts without really understanding what the script is doing or that they are even running a script.

The Windows PowerShell security features are intended to create a “secure by default” environment in which users cannot *easily* or *unknowingly* run scripts. This is not to say that the shell makes it impossible for users to run scripts because it does not. Rather, the shell makes it difficult—*by default*—for users to run scripts without realizing they are doing so.

The shell’s default security configuration is rarely convenient for someone who wants to run scripts frequently, and so the shell can be reconfigured to make script execution more convenient. This reconfiguration does make it somewhat easier for malicious scripts to enter the environment, and so the shell offers a range of security settings that let you strike the balance you want between convenience and security.

## Security Features

- No executable file type association.
  - The .ps1 filename extension for PowerShell scripts is not registered in Windows as an executable file type.
- No default local search.
  - The shell will not search the current path for script files. The "./" prefix is required to execute a script in the current folder.
- Restricted execution policy.
  - PowerShell's execution policy determines what scripts are permitted to run. Its default Restricted setting disables script execution entirely.

**PowerShell's default security settings are indeed restrictive.**

**However, do not modify these settings until you have considered the downsides of doing so.**

## Key Points

The shell offers three core security features related to scripts:

- The .ps1 filename extension used to identify Windows PowerShell scripts is not registered with Microsoft Windows® as an executable file type. By default, double-clicking a .ps1 file does not run the script (although it may open it in an editor such as Windows Notepad or the Windows PowerShell ISE).
- The shell does not search the current path for script files. Thus, if you type *myscript* into the shell, it does not execute the *myscript.ps1* file that may be located in the current directory. Instead, you would need to specify either an absolute or a relative path—such as *./myscript*—to the script. This behavior helps to prevent a form of attack called command hijacking, where a script executes instead of an internal command that has the same name.
- The shell has a script Execution Policy that determines what scripts are permitted to run, and by default this setting is set to Restricted, which disables script execution entirely.

It is very important that you not modify the shell's security features until you have fully considered the potential downsides of doing so. Your organization should adopt policies for Windows PowerShell script security to help ensure a consistently applied security policy that meets your operational and security needs.

## Execution Policy

- Five execution policy settings:
  - Restricted: Default setting, no script execution
  - RemoteSigned: Script execution enabled, remote scripts must be digitally signed
  - AllSigned: Script execution enabled, all scripts must be digitally signed
  - Unrestricted: Script execution enabled without restrictions
  - Bypass: Script execution enabled via bypassing policy entirely, intended for use where alternate security is in place
- Three methods for modifying the execution policy:
  - Group Policy: Uses a downloadable administrative template
  - Administrators: Running the Set-ExecutionPolicy cmdlet
  - Users: Using a command-line parameter with powershell.exe

### Key Points

The shell's execution policy can be changed in one of three ways:

- **By Group Policy.** A downloadable Group Policy administrative template is available from <http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=2917a564-dbdc-4da7-82c8-fe08b3ef4e6d>; this template is built into Windows Server 2008 R2. Any execution policy set via Group Policy overrides any locally configured setting.
- **By Administrators.** Running the **Set-ExecutionPolicy** cmdlet enables an administrator to change the system-wide execution policy on a given computer, provided the computer is not subject to a Group Policy setting as described previously. The systemwide execution policy is stored in the HKEY\_LOCAL\_MACHINE registry hive, which is normally writable only by local Administrators of the computer.
- **By Users.** Users can execute Windows PowerShell using powershell.exe and a command-line parameter that changes the execution policy for the duration of that shell session. This action overrides any local setting.

It may seem odd to permit users to override an administrator-established value for the execution policy, but remember that the execution policy is intended to help stop *unintended* script execution. It is not intended to stop skilled users from executing scripts at all, merely to ensure that they do not do so without knowing what they are doing. Running Powershell.exe and specifying a command-line parameter is definitely not something that can easily be accomplished by accident.

There are five settings for the execution policy:

- **Restricted.** This is the default setting, and Windows PowerShell does not execute scripts, except for a few Microsoft-provided, digitally-signed scripts that contain shell configuration defaults.

- **RemoteSigned.** This setting allows any script to be executed. However, remote scripts—those executed from a network location, those downloaded from the Internet using Internet Explorer, or those received in e-mail in Microsoft Office Outlook—must carry an intact, trusted digital signature.
- **AllSigned.** This setting allows any script to execute provided it carries an intact, trusted digital signature.
- **Unrestricted.** This setting allows any script to execute.
- **Bypass.** This setting bypasses the execution policy entirely, allowing any script to execute. This setting is primarily intended for developers who embed the shell inside another application, where the developer plans to provide their own security model rather than using the shell's own.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Configuring Script Security

- Learn how to configure script security

**Note:**  
**Please do not modify the script security settings in your virtual machines until instructed to do so in a lab**



### Key Points

In this demonstration, you will learn how to configure script security. However, please do not modify the script security settings in your virtual machines until instructed to do so in a lab.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod06\Democode\Configuring Script Security.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Execution Policy is Not Anti-Malware

- The execution policy is not intended to stop a determined user from executing a script
  - Remember that users can modify the execution policy as they invoke powershell.exe
- The execution policy is intended to assist users from unknowingly running a script
  - Extra effort is required to run any script in most policy configurations

**Users who run scripts still require the necessary privileges to accomplish the script's action**

**For example, adding or deleting users requires the necessary Active Directory privileges**

### Key Points

A determined user cannot be stopped from running a Windows PowerShell script simply by setting the execution policy. That is not the purpose of the execution policy.

Nor is the execution policy intended as a form of anti-malware. The execution policy is intended only to help prevent users from *unknowingly* running a script and to help users identify scripts that might be considered "trusted" and possibly safer to run.

Keep in mind that no user can use a shell script to perform some task for which they do not have permission. In other words, a normal non-administrator couldn't run a script to delete every user in Active Directory because the user wouldn't have the necessary permissions to do so. Simply being able to run a script doesn't change what someone can do with that script.

## Trusted Scripts

- A trusted script is one that has been digitally signed with a certificate that is trusted by your computer
  - Requires a Certification Authority
  - Requires a digital certificate
  - Requires a chain of trust
  - Requires the RemoteSigned or AllSigned execution policy

```
<!-- SIG # Begin signature block -->
<!-- MIIXXAYJKoZIhvCNQCCoIIXTTCCF0kCAQEcxAJBgUrDgMCggUAMGkGCisGAQQB -->
<!-- gjcCAQSwWzBZMDQGcIsGAQQBgjcCAR4wJgIDAQABAfzDtgvWUsTTrckOsYpVNR -->
<!-- AgEAAqEAqEAAqEAAqEAMCEwCQYFKw4DAhoFAAQUVJLcJAz9Hb6Ptn9NeoVABIG -->
<!-- fvKghhIxMIIEYDCCAOygAwIBAgIKLqsR3FD/XJ3LwDAJBgUrBgmCHQUAMHAxKzAp -->
<!-- BgNVBAsTIkNvchlyAwdodCaoYkgMTk5NyBnawNvby3NvznQgQ29ycC4xHjAcBgNV -->
<!-- BAsTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjehMBBGAlUEAxMYTwIJcm92b2Z0IFJv -->
<!-- b3QgQXVa0g9yaXR5MB4XD TA3MD gyMjIyMzEwMloXD TEyMD gyNTA3MD AwMFowETEL -->
<!-- MAKGA1UEBhMCVVMxEzARBgNVBAgTCIdhc2hpbmdb2x4EDAOBgNVBACTB1JlZG1v -->
<!-- bmQxhJaCgNvBAoTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjeJMCEGA1UEAxMaTWlJ -->
<!-- cm9zb2Z0IENvZGUgB2lnbmluZyBQQ0EwggiMA0GCSqGSIb3DQEBAQUAA4IBDwAw -->
<!-- ggEKoIBAQc3eX3WbNFOag0DHa+SU1XFa+x+ex0vx79FG6NSMw2tMUmILOmQLD -->
<!-- TdhJbCBKpmW/zio3C0i3f3xdRb2qjwS5qxSUrBqdNbDSMfOUEk+mKZzxFpZNKH5mN -->
<!-- syBiw0tfG/ZFR47DkQod29KIRmOyBMV/+J0imTwBh5z7urJf4L5XKCBhIW0 -->
<!-- sPK4IKPPOKZQhRcnh07dMPYAPFTG+T2BvobtbDmnLJt2C6vCn1ikXhmnJhzDYaV -->
<!-- BsTzILIPe01jyyzZMKUz7rtKljtQuxJOZf5qq2HyFY+n4JQIG4FsTXBey89UmY9 -->
```

## Key Points

Two of the shell's execution policy settings—RemoteSigned and AllSigned—utilize digital certificates and trust to identify "trusted" scripts.

So what is trust?

Trust starts with a *root Certification Authority*, or root CA. There are public CAs, and there are private companies that provide such certificates as a service, for a cost. Many companies also have their own private root CAs. When you say that you "trust a CA," you are saying that you have reviewed that CA's policies for verifying the identity of the people or companies to whom the CA issues digital certificates.

A *digital certificate* is a form of digital identity card. A digital certificate attests to the actual identity of a person or company, but that attestation is only as good as the trust you place in the company that issued the certificate. In other words, if a company contacts you online and claims to be Microsoft Corporation, you might look at their digital certificate. If their certificate also claims that the company is Microsoft Corporation, you would look at the issuer of that certificate to see whether you trust them to have researched the company's identity very thoroughly before issuing that certificate.

If you trust the CA to have done a good job of verifying the company's identity, you also trust the certificate they issued to that company. If that certificate is used to digitally sign a script, you also trust that script. That doesn't mean the script is beneficial or even free of malware; it simply means that you can identify the script's signer, and an intact signature ensures that the script was not modified because it was signed.

A digital signature is made using the encryption keys that are part of a digital certificate. The signature includes information about the certificate, including the identity of the certificate holder. The signature also contains encrypted information that can be used to verify the script's contents. If the signature and the contents of the script match, you know who signed the script, and you know that the script is exactly the same as it was when they signed it. Again, that does not mean the script is harmless—but if it turned out to be malicious, you could use the certificate information from the signature to track down the signer.



If you want to see what a digital signature looks like, run this command in Windows PowerShell:  
type \$pshome/types.ps1xml. The gibberish at the end of the file is the digital signature. You can  
check the status of the file's signature by running get-authenticodesignature  
\$pshome/types.ps1xml.

MCT USE ONLY. STUDENT USE PROHIBITED

## Signing a Script

- The Set-AuthenticodeSignature cmdlet
  - Is used to apply a digital signature to a script
  - Requires a Class 3 “code signing” digital certificate from a public CA or a local CA with a fully trusted certification path
  - Requires a Microsoft AuthentiCode style certificate
- Class 3 code signing certificates are commonly more expensive than Class 1 certificates for email signing
  - They usually require more stringent identity verification
  - However, one certificate can be used to sign every script
- A local CA can generate code signing certificates at no cost, but its certification path must be trusted by every client
  - Establishing this trust path is an extra administrative burden

### Key Points

To sign a script, you first need to obtain a trusted digital certificate.

One way to do obtain one is from a public or private CA. Public CAs generally charge a yearly fee for certificates. Private CAs are ones owned by your company and may be based on Windows Certificate Services or another certificate-management product.

The type of certificate you need is a Class 3 certificate, also known as a code signing certificate. From a public CA, these are commonly more expensive than the Class 1 certificates used to encrypt or sign e-mails, and these also require usually more stringent identity verification. Many CAs offer different variants of code signing certificates; you need a Microsoft AuthentiCode style certificate.

You can also generate a locally trusted certificate using the MakeCert.exe tool, which is available as part of the Windows Platform SDK. In Windows PowerShell, run this command to learn about MakeCert.exe and how to use it:

`Help about_signing`

A locally trusted certificate is trusted *only* by your local computer. Scripts signed using this kind of certificate are trusted for execution only on your local computer.

After you have installed a certificate, you use the Set-AuthenticodeSignature cmdlet to apply a signature to a script. The help file for this cmdlet contains details on how to use it, along with usage examples.



Read the help for Set-AuthenticodeSignature, and review some of its options. Can you find an example of how to use it to sign a script?

## Discussion: Selecting an Execution Policy

- What execution policy might be appropriate for your environment?
- Why?



### Key Points

In practical use, the RemoteSigned execution policy is useful because it assumes that local scripts are ones that you create yourself, and you trust them. It does not require those scripts to be signed. Scripts that are downloaded from the Internet or received via e-mail, on the other hand, are not trusted unless they carry an intact, trusted digital signature. You could certainly still run those scripts—by running the shell under a lesser execution policy, for example, or even by signing the script yourself—but those are additional steps you have to take, so it is unlikely that you would be able to run such a script accidentally or unknowingly.

The AllSigned execution policy is useful for environments where you do not want to accidentally run *any* script unless it has an intact, trusted digital signature. This policy is less convenient because it requires you to digitally sign every script you write, and re-sign each script each time you make any changes to it.

**Note:** Some third-party Windows PowerShell script editors can automatically sign your scripts for you, making the process more transparent and less inconvenient.

The Restricted execution policy is perfect for any computer for which you do not run scripts or for which you run scripts only rarely. (Keep in mind that you could always manually open the shell with a less-restrictive execution policy.)

The Unrestricted execution policy is not usually appropriate for production environments because it provides little protection against accidentally or unknowingly running untrusted scripts.

What execution policy might be appropriate for your environment? Why?

## Demonstration: Signatures and CAs

- Learn how to configure trusted CAs and work with signed scripts



### Key Points

In this demonstration, you will learn how to configure trusted CAs and work with signed scripts.

### Demonstration steps:

- 1 Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod06\Democode\Signatures and CAs.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 2

# Basic Scripts

- Create a basic script using one or more commands that have been tested interactively in the shell
- Execute a script from within the shell



A basic script is little more than a text file containing one or more commands to be executed in a sequence. At its simplest, you can create a basic script by copying and pasting commands from the command line into a text file.

### Lesson Objectives

After completing this lesson, you will be able to:

- Create a basic script using one or more commands that have been tested interactively in the shell.
- Execute a script from within the shell.

MCT USE ONLY. STUDENT USE PROHIBITED

## What Are Scripts?

- Scripts are text files with a .ps1 extension
  - Scripts contain one or more commands that you want the shell to execute in order
- Editing scripts can be accomplished in any text editor; however, the PowerShell ISE is particularly suited for script creation
  - Writing scripts within the shell is not a good practice
- Numerous third-parties offer alternate commercial and free editing environments
  - Idera's PowerShellPlus
  - iTripoli's AdminScriptEditor
  - SAPIEN Technologies' PrimalScript
  - Quest Software's PowerGUI

### Key Points

Scripts are text files that have a .ps1 filename extension. These files contain one or more commands that you want the shell to execute in a particular order.

You can edit scripts in Windows Notepad, but the Windows PowerShell ISE provides a better editing experience. In it, you can type commands interactively, immediately see the results, and paste those results into a script for long-term use. Or you can type your commands directly into a script, highlight each command, and press F8 to execute only the highlighted command. If you're pleased with the results, you simply save the script and you're done.

Third-party software vendors offer various commercial and free editing environments that seek to provide a better scripting experience.

Generally speaking, there are very few important differences between what you can do in a script and what you would do on the command line. Commands work in the same way in a script, meaning a script can literally be created by pasting commands that you have already tested at the command line.

## The Easy Way to Script

- Scripts are usually created when a PowerShell command needs to be used repeatedly. Such scripts often start as PowerShell commands in the shell.
- Three techniques for converting PowerShell commands into reusable scripts:
  - Run commands in the console window, and then copy and paste those commands into a script.
  - Run commands in the Windows PowerShell ISE, and then copy and paste those commands into a script.
  - Write commands directly in a script in the Windows PowerShell ISE, and test commands by highlighting them and pressing F8. When you have the command working properly, it's already in a script and ready to be saved.
- The reverse of any of these techniques is also a useful administrative technique.

### Key Points

Scripting is often easier when you can assemble a script from already-tested commands. There are three techniques:

- Run commands in the console window, and then copy and paste those commands into a script.
- Run commands in the Windows PowerShell ISE, and then copy and paste those commands into a script.
- Write commands directly in a script in the Windows PowerShell ISE, and test commands by highlighting them and pressing F8. When you have the command working properly, it's already in a script and ready to be saved.

If you are working with a third-party script editor, it may offer similar capabilities for quickly moving from the command line to a script.

Note that the reverse process is also true: You can copy a command from a script and paste it into the command line to run that command. Of course, in the Windows PowerShell ISE, you can simply highlight a command and press F8 to execute just that command.

However, scripts also make it easy to add some documentation to your commands. As you add commands to a script, also add comments to help document what each command is doing. That way, someone else reading the script—or even yourself, a few months later—will have fewer troubles figuring out what the commands were supposed to be doing.

To create a comment in a script, simply type the pound sign, followed by your comment:

```
# Retrieve WMI object for the operating system  
Get-WmiObject Win32_OperatingSystem
```

## Improving Script Readability

- Because scripts are intended for long-term use, it is important to write them clearly
- Three best practices for script readability:
  - Always use full cmdlet names, not aliases, in a script
  - Always use full parameter names, not truncated versions, in a script
  - Break long lines into several physical lines by pressing Return after the pipe character

```
Get-Process |  
Sort-Object VM -descending |  
Select-Object -first 10 |  
Export-Csv procs.csv
```

### Key Points

Commands on the command line aren't always pretty or easy to read. For example, this is a valid, if difficult-to-read, command:

```
Gwmi win32_operatingsystem -comp (gc names.txt) | % { $_.Reboot() }
```

At the command line, you may not care about readability—your goal is to minimize typing, which is what things like aliases and truncated parameter names provide.

Scripts, however, are meant for longer-term use, and they are easier to use and maintain if they are easier to read. Here are some best practices to follow:

- Always use full cmdlet names, not aliases, in a script.
- Always use full parameter names, not truncated versions, in a script.
- Break long lines into several physical lines by pressing Return after the pipe character.

This last practice can be especially helpful for making the script easier to read. For example, rather than this command:

```
Ps | sort vm -desc | select -fir 10 | export-csv procs.csv
```

You might create a script like this:

```
Get-Process |  
Sort-Object VM -descending |  
Select-Object -first 10 |  
Export-Csv procs.csv
```

The indentation on the second, third, and fourth lines helps to connect those visually to the cmdlet that started the command line.

Note than some shell users use the escape character to escape the carriage return and act as a line continuation character:

```
Get-Process | ` 
Sort-Object VM -descending | ` 
Select-Object -first 10 | ` 
Export-Csv procs.csv `
```

This technique works, but it does make the script more difficult to read because the escape character—a backtick—is difficult to discern in some fonts.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Writing Basic Scripts

- Review how to write a basic script



### Key Points

In this demonstration, you will review how to write a basic script.

### Demonstration steps:

- 1 Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod06\Democode\Writing Basic Scripts.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Making It Easier to Make a Script

- Running PowerShell commands from within the PowerShell ISE makes it easy to shift from “commands” to “scripts”
  - Typing a command into an ISE script window means that you are already typing a script
  - No need to copy and paste completed commands to create scripts. Simply save the file
  - The ISE supports a broader variety of character sets
- The ISE executes commands in much the same way as the console window
  - Be aware that cmdlets like Read-Host generate a graphical dialog input box rather than a command -ine prompt
  - Scripts that you plan to execute as scheduled tasks should be tested in the console window prior to installation

### Key Points

If you work primarily in the Windows PowerShell ISE (or a similar third-party script editor), you can move from *command* to *script* much more efficiently. Because you are already typing a script, there’s no need to copy and paste completed commands—just save the file.

The ISE also supports a broader variety of character sets, including double-byte character sets needed for some languages. For users working in a double-byte character set language, the ISE is far superior to the console window.

The ISE executes commands in much the same way as the console window, although cmdlets like Read-Host generate a graphical dialog input box rather than a command-line prompt. If you plan to write a script that will execute as a scheduled task, you should test it in the console window after writing it in the ISE, just to be sure everything works as expected.

**Note:** If you have not already been doing so, start to use the Windows PowerShell ISE rather than the console window. The ISE has a minimal graphical user interface and it should take very little additional time to become familiar with it.

## Lesson 3

# Parameterized Scripts

- Identify elements of a script that may need to change each time the script is executed
- Replace variable elements with parameters
- Execute a parameterized script from within the shell using named and positional parameters



Sometimes, you want to run a script with variable information; you want to be able to change a computer name or filename. One way to accomplish this is to edit the script each time you run it, entering the information you want each time. But parameterized scripts offer a way to include variable information in your script without having to edit it every time.

### Lesson Objectives

After completing this lesson, you will be able to:

- Identify elements of a script that may need to change each time the script is executed.
- Replace variable elements with parameters.
- Execute a parameterized script from within the shell using named and positional parameters.

## Why Parameterize?

- Scripts that contain hard-coded values are more difficult to reuse in a variety of situations
  - Specified computer names
  - Specified IP addresses
  - Specified file names
  - Localization
- Using parameters enables you to formally structure the variable parts of your script into the script itself
  - This process enables the script to be run, without your needing to edit it for each unique use
  - Changeable information is then provided each time the script is run
  - Parameterization is similar to how command-line utilities and cmdlets require information at their time of execution

### Key Points

When scripts contain hard-coded values, such as computer names, or IP addresses or filenames, they are more difficult to reuse in a variety of situations, and they are more difficult for less-skilled shell users to use.

Having to edit a script each time you run it opens up more potential for errors. If you're providing the script to a less-skilled technician, that person might edit portions of the script that you didn't intend, creating more work and confusion.

Using parameters enables you to have the variable parts of your script formally structured into the script itself. The script can be run without having to be edited and the changeable information provided each time the script is run—much like a command-line utility or cmdlet.

## Discussion: What Would You Parameterize?

- What kinds of information might go into a script that should be parameterized?
  - Computer names
  - IP addresses
  - User names
  - Domain names
  - What else?



### Key Points

What kinds of information might go into a script that should be parameterized?

Computer names, IP addresses, user names, domain names, and so forth—what else?

## Declaring Parameters

- The Param() block
  - Is used to declare parameters in a script
  - Uses parenthesis, not curly braces
  - Should be placed near the top of your script. It must be before any commands, but can be after script comments
- Parameter names start with a \$, and generally consist of letters, numbers, and the \_ character
  - Parameter names with spaces should be enclosed with {}
  - It is considered a good practice to avoid spaces
  - Parameters in a Param() block should be separated by commas

```
Param (  
    $computerName = 'localhost',  
    $userName  
)
```

### Key Points

Parameters are declared in a Param() block near the top of your script. This block must appear before any of the commands in the script, although it can appear after comments that explain what the script does.

**Note:** It's a good idea to include comments explaining what each parameter is, and what sort of values each parameter expects.

After they are declared, the parameters act as placeholders and can be used anywhere that you would normally use a hard-coded value.

Parameter names start with a dollar sign (\$) and generally consist of letters, numbers, and the underscore character. You can have parameter names that contain spaces if you enclose the name in curly braces:

```
 ${My Parameter}
```

However, this technique does make the script a bit less readable, so most Windows PowerShell script authors try to avoid spaces in the parameter names.

Here is an example:

```
Param (  
    $computerName,  
    $userName  
)
```

Note that the parameter list is comma-separated, and you can improve readability by listing each parameter on its own line. If you do so, however, don't forget to include the comma after every parameter name except the first one.

Also note that the Param() block uses parentheses, not curly braces.

You can also provide default values for a parameter:

```
Param (
    $computerName = 'localhost',
    $userName
)
```

This way, if the script is run without a value provided for the \$computerName parameter, the script can still run using the default 'localhost' value. Parameters without a default value default to an empty string or zero if the script is run without that parameter.

## Using Parameters

- Parameterized scripts can use positional parameters or named parameters

- This parameter block:

```
Param (  
    $computerName,  
    $userName  
)
```

- Can be executed using this command line with positional parameters:

```
./myscript 'localhost' 'Administrator'
```

- Or using this command line with named parameters:

```
./myscript -computerName 'localhost' -userName 'Administrator'
```

### Key Points

When you run a parameterized script, you can provide parameters either positionally or by using names. If a script named MyScript.ps1 contains this parameter block:

```
Param (  
    $computerName,  
    $userName  
)
```

you can run it like this:

```
./myscript 'localhost' 'Administrator'
```

or like this, which is easier to read:

```
./myscript -computerName 'localhost' -username 'Administrator'
```

If you use the positional technique, separate parameter values with spaces, not commas. If you use the named technique, parameters can appear in any order, and parameter names may be truncated as they are in cmdlets.

## Prompting

- For some scripts, it may be useful to prompt the script's user for a value
  - Further prompting is also handy when values are not provided for a parameter
- The Read-Host cmdlet
  - Enables the creation of a prompt for input at script execution

```
Param (  
    $computerName = $(Read-Host 'Enter computer name'),  
    $userName = $(Read-Host 'Enter user name')  
)
```

## Key Points

Sometimes, you might wish to prompt a script's users for a value or prompt them only if they do not provide a value for a parameter. The Read-Host cmdlet enables you to do this. For example, to prompt someone for a parameter that was not provided, use Read-Host as part of the default value:

```
Param (  
    $computerName = $(Read-Host 'Enter computer name'),  
    $userName = $(Read-Host 'Enter user name')  
)
```

You can learn more about Read-Host by reading its help file within the shell.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Writing Parameterized Scripts

- Review how to parameterize a script



### Key Points

In this demonstration, you will review how to parameterize a script.

### Demonstration steps:

- 1 Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod06\Democode\Writing Parameterized Scripts.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab: Writing Windows PowerShell Scripts

- Exercise 1: Executing Scripts
- Exercise 2: Using Positional Script Parameters
- Exercise 3: Using Named Script Parameters

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are a system administrator, and have commands that you run regularly at work. To save time and also to be able to more easily pass around your commonly used commands to others, you have decided to copy your commands into a script. To make it easier for more junior administrators or even desktop users to use your script, you've added simple command-line arguments that can be passed dynamically to your script when run.

#### Lab Setup

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

- 1 Start the **LON-DC1** virtual machine. You do not need to log on but wait until the boot process is complete.
- 2 Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 3 Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Executing Scripts

### Scenario

To administer your systems, you like to copy commonly used commands into a script so you can easily invoke them later. The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Check the execution policy setting.
2. Create a script.
3. Execute a script from the shell.

► **Task 1: Check the execution policy setting**

- Check the execution policy to make sure scripts can be executed without needing to be signed.

► **Task 2: Create a script**

- Create a PowerShell script **C:\Script.ps1** that contains a command will get all the running *powershell* processes. You can use Notepad to do this then replace the file extension from **.txt** to **.ps1** once you are finished.

► **Task 3: Execute a script from the shell**

- Run the script from the PowerShell console calling the script name **C:\Script.ps1**.

**Results:** After this exercise, you will have created and run a simple script that lists all the "powershell" processes.

## Exercise 2: Using Positional Script Parameters

### Scenario

With basic scripting firmly understood, you now want to expand a few scripts to make them easier to use. You want to be able to easily pass parameters to your script so that they can be more generalized.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Copy commands into a script.
2. Use commands with hard-coded values as parameter values.
3. Identify variable portions of a command.
4. Create positional script parameters.
5. Execute a script from the shell.

#### ► Task 1: Copy commands into a script

- Create a script named **C:\Script2.ps1** containing the following command:

```
Get-EventLog -LogName Application -Newest 100 | Group-Object -Property InstanceId | Sort-Object -Descending -Property Count | Select-Object -Property Name,Count -First 5 | Format-Table -AutoSize
```

#### ► Task 2: Use commands with hard-coded values as parameter values

- Run the script just created.

#### ► Task 3: Identify variable portions of a command

- Review the script and determine what parts of the command could be considered variable values.

#### ► Task 4: Create positional script parameters

- Create a new script **C:\Script3.ps1** that accepts positional script parameters for the four values underlined below.

**Hint:** Create a script that accepts four parameters, such as \$log, \$new, \$group, and \$first. In addition to defining those parameters in a Param() block, use those parameters in place of the hard-coded values in the command.

```
Get-EventLog -LogName Application -Newest 100 | Group-Object -Property InstanceId | Sort-Object -Descending -Property Count | Select-Object -Property Name,Count -First 5 | Format-Table -AutoSize
```

#### ► Task 5: Execute a script from the shell

- Run the script just created by passing back the same values that were replaced by the parameters (in the same order).

**Hint:** To run the script, provide the four parameterized values, - such as "Application," 100, "InstanceId," and 5, in the order in which they are defined in the Param() block.

**Results:** After this exercise, you will have created and run a script that uses positional parameters instead of hard-coded values.

## Exercise 3: Using Named Script Parameters

### Scenario

You want to be able to easily pass parameters to your script so that they can be more generalized. This time, however, you wish to use named parameters as opposed to positional parameters in your script.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Copy commands into a script.
2. Use commands with hard-coded values as parameter values.
3. Identify variable portions of a command.
4. Create named script parameters.
5. Execute a script from the shell.

#### ► Task 1: Copy commands into a script

- Create a script **C:\Script4.ps1** that contains the following command:

```
Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2 -  
MaxSamples 3
```

#### ► Task 2: Use commands with hard-coded values as parameter values

- Run the script just created.

#### ► Task 3: Identify variable portions of a command

- Review the script and determine what parts of the command could be considered variable values.

#### ► Task 4: Create named script parameters

- Create a new script **C:\Script5.ps1** that accepts named script parameters for the three values underlined below.

**Hint:** Create a Param() block with three variables, such as \$cpu, \$interval, and \$max. Then, use those variables in place of the hard-coded values \_Total, 2, and 3.

```
Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2 -  
MaxSamples 3
```

Do not run this script now. You will run it in the next task.

#### ► Task 5: Execute a script from the shell

- Run the script that you just created. When you do so, pass in the same values that were just replaced by parameters by specifying the parameter names and the corresponding values.

**Hint:** You need to provide values for the parameters you just added to the script. If your parameters were \$cpu, \$interval, and \$max, you run the script by using the -cpu, -interval, and -max parameters, passing appropriate values to each.

**Results:** After this exercise, you will have created and run a script that uses named parameters instead of using hardcoded values.

## Lab Review

1. If you send a script to another user with Windows 7 or Windows Server 2008 R2, will that person be able to run the script? Is PowerShell included with these versions? Is it included with older Windows operating system versions?
2. If you send a script to another user, how can that person incorporate the script so that it automatically runs every time PowerShell is opened?
3. Is there a way to specify a default value for parameters in the param statement?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- What execution policy requires that remote scripts have a digital signature?
- What are the advantages of a parameterized script?

### Class Discussion

- Common issues related to scripting



### Review Questions

1. What execution policy requires that remote scripts have a digital signature?
2. What are the advantages of a parameterized script?

### Common Issues Related to Scripting

Identify the causes of the following common issues related to the core cmdlets, and fill in the troubleshooting tips. For answers, refer to relevant lessons in the module.

Issue	Troubleshooting tip
Script does not execute because the execution policy prohibits execution of scripts.	

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 7

## Background Jobs and Remote Administration

### Contents:

<b>Lesson 1:</b> Working with Background Jobs	7-3
<b>Lab A:</b> Working with Background Jobs	7-15
<b>Lesson 2:</b> Using Windows PowerShell Remoting	7-24
<b>Lab B:</b> Using Windows PowerShell Remoting	7-56

## Module Overview

After completing this module, you will be able to:

- Start and manage background jobs
- Receive results from background jobs
- Integrate background jobs into other tasks and processes
- Manage remote computers by using one-to-one remoting
- Execute commands on multiple remote computers
- Import remote commands into the local shell
- Use implicit remoting to run commands that are located on a remote computer



Background jobs and remoting are two key features introduced in Windows PowerShell® v2.0. These features each add a level of maturity to the administrative capabilities of Windows PowerShell, helping you to accomplish more complex tasks and enabling you to extend an administrative reach across your enterprise.

### Module Objectives

After completing this module, you will be able to:

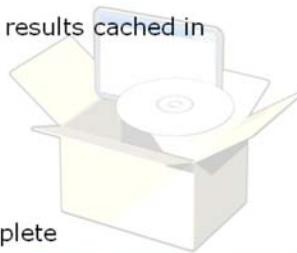
- Start and manage background jobs.
- Receive results from background jobs.
- Integrate background jobs into other tasks and processes.
- Manage remote computers by using one-to-one remoting.
- Execute commands on multiple remote computers.
- Import remote commands into the local shell.
- Use implicit remoting to run commands that are located on a remote computer.

## Lesson 1

# Working with Background Jobs

After completing Lesson 1, you will be able to:

- Explain how background jobs are executed in the shell
- Create a new background job for WMI operations
- Create a new background job that executes commands on the local computer
- Display current job status
- Retrieve the results from a job and work with those results in the pipeline
- Retrieve the results from a job and leave the results cached in memory
- Retrieve the properties of a job
- Remove a completed job
- Wait for a job to complete
- Stop a job that has hung or that will not complete



When executing some long-running commands in the shell, you may prefer to have them take place in the background, so that you can continue using the shell for other tasks.

One way to accomplish this would be to simply open a second shell window, but that involves using a bit more memory and processor time on your computer and gives you another window to manage. Another option is to have the commands run as a background job.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain how background jobs are executed in the shell.
- Create a new background job for WMI operations.
- Create a new background job that executes commands on the local computer.
- Display current job status.
- Retrieve the results from a job and work with those results in the pipeline.
- Retrieve the results from a job and leave the results cached in memory.
- Retrieve the properties of a job.
- Remove a completed job.
- Wait for a job to complete.
- Stop a job that has hung or that will not complete.

MCT USE ONLY. STUDENT USE PROHIBITED

## Interactive vs. Background

- Most PowerShell commands complete interactively, also known as *synchronously*
  - The first command must complete before the next command can start
  - This works well when subsequent commands rely on data from previously-run commands
- Sometimes you want a command or series of commands to run in the background
  - Background jobs move processing to the background
  - Commands execute, but the shell is freed for other activities
- Results from background jobs are saved in memory as a job object
  - Job object data can be retrieved from completed jobs
  - Job object data can be used for other background or interactive commands

### Key Points

Normally, Windows PowerShell commands complete interactively, or *synchronously*. If you execute a long-running command, you have to wait for it to complete before running other commands in the shell.

Background jobs simply move the command processing into the background. The command continues to execute, but you can begin using the shell for other tasks. Windows PowerShell provides cmdlets for checking on the status of a background job.

When a command runs in the background, its results are saved in memory as a part of a *job object*. You can retrieve the results of completed jobs and work with those results in the pipeline.

## Background WMI

- The Get-WmiObject cmdlet...

- Often requires a very long time to run. This is particularly the case when Get-WmiObject is run against multiple remote computers at once.
- Has an –AsJob parameter, which runs its query as a background job.

```
Get-WmiObject -computer (gc names.txt) -class  
CIM_DataFile -AsJob
```

- Rather than waiting for WMI to complete, the –AsJob parameter instead places a job object in the pipeline and begins executing in the background.
- The –AsJob parameter can be found in other cmdlets, some of which will be discussed in this module.

### Key Points

Get-WmiObject is one cmdlet that often takes a long time to run, especially when you are retrieving information from multiple remote computers. The cmdlet supports an –AsJob parameter, which runs the cmdlet as a background job. You use Get-WmiObject in exactly the way you normally would; simply specify

–AsJob as an additional parameter to create a background job:

```
Get-WmiObject -computer (gc names.txt) -class CIM_DataFile -AsJob
```

Rather than waiting for WMI to complete, the command will immediately place a job object into the pipeline and begin executing the job in the background.

**Note:** Other cmdlets may also offer an –AsJob option. A job is an extension point in Windows PowerShell, meaning developers of cmdlets can create a new and different kind of job object. The cmdlets discussed in this lesson all produce the same kind of job object, and so all of the job management techniques you learn will apply to those jobs. Jobs created by other third-party cmdlets, or even by other Microsoft product teams, may differ somewhat.

## Managing Jobs

- Four important cmdlets are used to manage the list of running jobs
  - Get-Job
  - Remove-Job
  - Wait-Job
  - Stop-Job

**These management cmdlets should feel similar to the command line tools you've used in the past for at jobs and scheduled tasks**

### Key Points

Once a job is running, you can monitor its progress, remove the complete job, or retrieve the results from a completed job. The shell provides several job management cmdlets:

- **Get-Job** will retrieve all available jobs. Specify the –id or –name parameter to retrieve a specific job. Pipe this command's output to Get-Member or to Format-List \* to see the properties of a job.
- **Remove-Job** will remove a job. Specify either the –id or –name parameter to specify which job, or pipe jobs into this cmdlet to remove them.
- **Wait-Job** will pause and wait for a job to complete. Specify either the –id or –name parameter to specify which job, or pipe a job into this cmdlet to wait for the job. This cmdlet can be useful in scripts, when you need the script execution to pause until a given background job has completed.
- **Stop-Job** will halt the execution of a job, including jobs that have hung and will not complete on their own. Specify either the –id or –name parameter to specify which job, or pipe a job into this cmdlet to stop the job.

## Demonstration: WMI Jobs

- Learn to start jobs by using Get-WmiObject



### Key Points

In this demonstration, you will learn to start jobs by using Get-WmiObject.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\WMI\Jobs.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Job Output

- At some point, you need to gather the data from a background job. That data is contained within the job's Job Object.
- The **Receive-Job** cmdlet...
  - Obtains the results from a background job in memory.

```
Receive-Job -id 2 | Sort State | Where { $_.State -ne "Stopped" }
```

- It is also useful to assign the results of **Receive-Job** to a variable.

```
$results = Receive-Job -id 2
```

- Retrieving the output of a job also removes that output from the job's memory cache. Use the **-keep** parameter to retain this data for later use.

### Key Points

When a job completes, the output of its commands are cached in memory as part of the job object. You use the **Receive-Job** cmdlet to obtain those results from the memory cache.

**Receive-Job** places objects into the pipeline. Suppose you start a job by using this command:

```
Get-WmiObject Win32_Service -comp Server2 -asjob
```

And the resulting job looks like this:

Id	Name	State	HasMoreData
--	---	-----	-----
2	Job2	Completed	True

Then you could receive the results into the pipeline, filter them, and sort them, as follows:

```
Receive-Job -id 2 | Sort State | Where { $_.State -ne "Stopped" }
```

You could also assign the results of **Receive-Job** to a variable:

```
$results = Receive-Job -id 2
```

**Note:** You will learn more about variables in the module 11.

Note that the prior example actually follows a fairly poor practice. It is querying more information from WMI than is really needed, and then filtering that information locally. A better technique would be:

```
Gwmi Win32_Service -comp Server2 -filter "State <> 'Stopped'" -asjob
```

This example accomplishes the filtering at the source, following the best practice of *filter left*, meaning to place the filtering criteria as far to the left of the command line as possible.

When you receive the output of a job, the shell removes the output from the job's memory cache. That means you can only use Receive-Job one time against any given job; after doing so, there will be no other results to receive. By specifying the –keep parameter, however, you can instruct the shell to give you the job results and to retain them in the job's cache.

```
Receive-Job -id 4 -keep
```

Another way to retain the job results is to export them to an XML file:

```
Receive-Job -id 7 | Export-CliXML jobresults.xml
```

You can then use Import-CliXML to re-import the job results into the shell.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Job Output

- Learn how to work with job output



### Key Points

In this demonstration, you will learn how to work with job output.

### Demonstration steps:

- 1 Start the **LON-DC1**, **LON-SVR1** and **LON-SVR2** virtual machines and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Working with Job Output.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Managing the Job List

- The shell retains information about in-progress and completed jobs, even after you've received job results.
- You will need to clean up the job list from time to time, using the Remove-Job cmdlet. There are multiple approaches you can use:
  - By ID
  - By Instance ID
  - By name
  - By state
  - By piping jobs to Remove-Job

```
Remove-Job -id 1
Remove-Job -name Job1
Get-Job | Where { $_.Name -like "AD*" } | Remove-Job
```

### Key Points

The shell retains information about in-progress and completed jobs, even after you have received the results of the job.

You can clean up the job list by removing jobs that have been completed. Note that doing so removes any cached output—you won't be able to receive the job results after removing the job.

Specify jobs either by ID:

```
Remove-Job -id 1
```

By name:

```
Remove-Job -name Job1
```

Or by piping jobs to the cmdlet:

```
Get-Job | Where { $_.Name -like "AD*" } | Remove-Job
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Removing Jobs

- Learn how to manage the job list



### Key Points

In this demonstration, you will learn how to manage the job list.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Removing Jobs.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Other Commands As Jobs

- As mentioned before, some cmdlets other than Get-WmiObject can be converted to jobs using the `-AsJob` parameter
- Alternatively, any command can become a background job by using the **Start-Job** cmdlet
  - `Start-Job` executes a script block in the background
  - `Start-Job` can leverage alternate credentials

**Start-Job -scriptblock {Get-Process}**

- The **Invoke-Command** cmdlet can also run jobs in the background, but only against remote computers
  - `Invoke-Command` will be covered in this module's next lesson

### Key Points

Background jobs aren't limited to WMI. Other cmdlets support an `-AsJob` parameter, and you can use any of these cmdlets to start a new, local background job.

In addition, *any* command can become a background job by using the **Start-Job** cmdlet. Simply provide a script block (that is, one or more commands). You can optionally provide alternate credentials for the job to run under. `Start-Job` always starts a job, so there is no `-AsJob` parameter. There is, however, a `-Name` parameter, which enables you to specify a name for the job. This name will be used rather than an automatically-generated job name like `Job1` or `Job2`. Specifying your own job name makes it easier to refer to that job using the `-Name` parameter of cmdlets such as `Get-Job`, `Wait-Job`, `Remove-Job`, `Receive-Job`, and `Stop-Job`.

You may also notice that **Invoke-Command** has an `-AsJob` parameter. However, do not use this to start a local job, as it is intended to be used to start background jobs that communicate with remote computers. This cmdlet will be covered in the next lesson.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Local Commands As Jobs

- Learn how to run local commands as background jobs



### Key Points

In this demonstration, you will learn how to run local commands as background jobs.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Local Commands as Jobs.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab A: Working with Background Jobs

- Exercise 1: Using Background Jobs with WMI
- Exercise 2: Using Background Jobs for Local Computers
- Exercise 3: Receiving the Results from a Completed Job
- Exercise 4: Removing a Completed Job
- Exercise 5: Waiting for a Background Job to Complete
- Exercise 6: Stopping a Background Job Before It Completes
- Exercise 7: Working with the Properties of a Job

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	ContosoAdministrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are a system administrator who has several scripts already developed. Some of the scripts can take anywhere from a few seconds to several minutes to complete. You would waste a lot of productive time if you ran the scripts one-by-one and waited for each to finish. To save time, you are going to use background jobs to run all the scripts synchronously, and be able to control everything from one PowerShell console.

#### Lab Setup

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

- 1 Start the **LON-DC1** virtual machine. You do not need to log on, but wait until the boot process is complete.
- 2 Start the **LON-SVR1** virtual machine and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 3 Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Using Background Jobs with WMI

### Scenario

You use WMI to gather a lot of information from servers and would like to use background jobs so you can continue using the console while they are running. The main tasks for this exercise are as follows:

1. Start a background job.
2. Check the status of a background job.
3. Use Windows Management Instrumentation.
4. Retrieve information from remote computers.

Perform these tasks on the LON-SVR1 virtual machine.

► **Task 1: Start a background job**

- Open **Windows PowerShell** and run a background job that simply runs **Get-Process -Name PowerShell**.

► **Task 2: Check the status of a background job**

- Check the status of the background job just started.

► **Task 3: Use Windows Management Instrumentation**

- Use WMI to retrieve the local computer name (use the **Win32\_ComputerSystem** class).

► **Task 4: Retrieve information from remote computers**

- Using a background job, run the previous WMI command against LON-DC1 and view the results.

**Results:** After this exercise, you will have used a background job that uses WMI to retrieve the Win32\_ComputerSystem class from LON-DC1 remotely.

## Exercise 2: Using Background Jobs for Local Computers

### Scenario

You often run commands that take a long time and would like to use background jobs so you can continue using the console while they are running.

The main tasks for this exercise are as follows:

1. Start a background job.
2. Check the status of a background job.
3. Retrieve information from local commands.

► **Task 1: Start a background job**

- Start a background job that retrieves all of the services on the local machine.

► **Task 2: Check the status of a background job**

- Check the status of the background job.

► **Task 3: Retrieve information from local commands**

- Retrieve the results from the background job.

**Results:** After this exercise, you will have used a background job to retrieve a listing of all the installed services on LON-SVR1.

## Exercise 3: Receiving the Results from a Completed Job

### Scenario

You often use background jobs to run commands and now would like to view the results of the jobs you have started and are completed. The main tasks for this exercise are as follows:

1. Run a background job.
2. Check the status of a background job.
3. Receive results from a background job and keep the results in memory.
4. Remove the results of a background job from memory.

Perform these tasks on the LON-SVR1 virtual machine.

► **Task 1: Run a background job**

- Start a background job that will retrieve the most recent 100 events from the **Windows Application** event log.

► **Task 2: Check the status of a background job**

- Check the status of the background job just started.

► **Task 3: Receive the results of a background job and keep the results in memory**

- Receive the results of the background job just started and keep the results in memory.

► **Task 4: Remove the results of a background job from memory**

- Receive the results of the background job just started and remove the results from memory.

**Results:** After this exercise, you will have used a background job to retrieve a listing of the most recent 100 events in the Windows Application event log. You also will have seen how to save and remove the results from memory.

## Exercise 4: Removing a Completed Job

### Scenario

You often use background jobs to run commands and now would like to remove some of the jobs from memory so the listing of any previously completed jobs is cleared up. The main tasks for this exercise are as follows:

1. Run a background job.
2. Check the status of a background job.
3. Remove a completed job.

Perform these tasks on the LON-SVR1 virtual machine.

- ▶ **Task 1: Run a background job**
  - Start a background job that will run the **Get-HotFix** cmdlet.
- ▶ **Task 2: Check the status of a background job**
  - Check the status of the background job just started.
- ▶ **Task 3: Remove a completed job**
  - Once completed, remove the completed job from memory.

**Results:** After this exercise, you will have run Get-HotFix as a background job, checked its status, and once completed, will have removed the entire job from memory.

## Exercise 5: Waiting for a Background Job to Complete

### Scenario

You often use background jobs to run commands and now would like to save some of the results to a text file for future reference. The main tasks for this exercise are as follows:

1. Start a background job.
2. Check the status of a background job.
3. Wait for background job to complete.
4. Copy the results of a background job to a file.

Perform these tasks on the LON-SVR1 virtual machine.

► **Task 1: Start a background job**

- Start a background job that simply runs the **Start-Sleep** cmdlet with a parameter value of **15 seconds**.

► **Task 2: Check the status of a background job**

- Check the status of the background job just started.

► **Task 3: Wait for a background job to complete**

- Run a command that will wait until the background job is completed.

► **Task 4: Copy the results of a background job to a file**

- Copy the results of the background job to a local text file.

**Results:** After this exercise, you will have run Start-Sleep for 15 seconds, checked its status, and will have run the Wait-Job cmdlet to ensure the job was completed. You will also have copied the results of the background job to a local file.

## Exercise 6: Stopping a Background Job Before It Completes

### Scenario

You often use background jobs to run commands and have noticed that one of your jobs appears to be hung. You would like to stop it to make sure it isn't using any server resources.

The main tasks for this exercise are as follows:

1. Start a background job.
2. Check the status of the background job.
3. Stop a background job that has not yet completed.

Perform these tasks on the LON-SVR1 virtual machine.

► **Task 1: Start a background job**

- Start a background job that runs as an infinite loop.

**Hint:** The script block for this job will be something like this:  
{ while (\$true) { Start-Sleep -seconds 1 } }

► **Task 2: Check the status of the background job**

- Check the status of the background job just created.

► **Task 3: Stop a background job that has not yet completed**

- Stop the background job even though it is still running.

**Results:** After this exercise, you will have run a simple endless command as a background job, checked its status, and will have stopped the background job before it returned a completed status.

## Exercise 7: Working with the Properties of a Job

### Scenario

You often use background jobs to run commands and would like to display some of the properties of those jobs.

The main tasks for this exercise are as follows:

1. Start a background job.
2. Display a list of background jobs.
3. Use **Get-Member** to look at the properties of a job.

Perform these tasks on the LON-SVR1 virtual machine.

► **Task 1: Start a background job**

- Start a background job that gets a listing of all the directories and files in **C:\Windows\System32**.

► **Task 2: Display a list of background jobs**

- Display a list of all the background jobs on the system.

► **Task 3: Use Get-Member to look at the properties of a job**

- Use **Get-Member** to look at the properties of the background job just created.

**Results:** After this exercise, you will have started a background job that retrieves a listing of all the contents of C:\Windows\System32, used Get-Member to view the methods and properties of a background job, and you will have displayed the ChildJobs property of the background job.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. You have started a PowerShell console and have started a couple of background jobs. What happens to the background jobs if they are still running and the main console is closed?
2. If you have two or more PowerShell consoles open, and start a background job in one console, can you access the job from the other console?

## Lesson 2

# Using Windows PowerShell Remoting

After completing this lesson, you will be able to:

- Explain the architecture of Windows Remote Management (WinRM)
- Configure WinRM on the local computer
- Configure WinRM in a domain environment
- Configure Windows Remote Shell in a domain environment
- Open, use, and close an interactive remote shell session
- Invoke a command that executes on one or more remote computer
- Work with the results of commands that executed on multiple computers
- Invoke a command that executes on multiple computers and that runs as a background job
- Receive the results of a background job that includes commands that executed on remote computers
- Create, manage, and remove reusable objects that represent connections to remote computers
- Use implicit remoting to execute commands that are installed on a remote computer

While many Windows PowerShell cmdlets—such as Get-WmiObject or Get-Service—already implement their own remote connectivity features, Windows PowerShell also provides its own *remoting* features that allow almost any command or script to be run on one or more remote computers. These features help extend your administrative reach to include any computer on your network that is running Windows PowerShell v2, and they enable you to manage multiple computers with a single command.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the architecture of Windows Remote Management (WinRM).
- Configure WinRM on the local computer.
- Configure WinRM in a domain environment.
- Configure Windows Remote Shell in a domain environment.
- Open, use, and close an interactive remote shell session.
- Invoke a command that executes on one or more remote computers.
- Work with the results of commands that executed on multiple computers.
- Invoke a command that executes on multiple computers and that runs as a background job.
- Receive the results of a background job that includes commands that executed on remote computers.
- Create, manage, and remove reusable objects that represent connections to remote computers.
- Use implicit remoting to execute commands that are installed on a remote computer.

## Why Manage Remotely?

- Now that your knowledge of command-line management is growing, it no longer makes sense to perform many tasks at the computer's console
  - Managing at the console is a poor management practice
  - Managing at the console consumes excess on-system resources, which can impact running processes
  - Managing at the console takes extra steps, which slows down your ability to solve problems and enact change
  - Managing at the console is not repeatable
  - Managing at the console has a greater possibility of error, especially when repeating complicated tasks
- Microsoft has gradually added remote management capabilities to its administrative tools
- With PowerShell, nearly any command can be executed on one or more remote computers

### Key Points

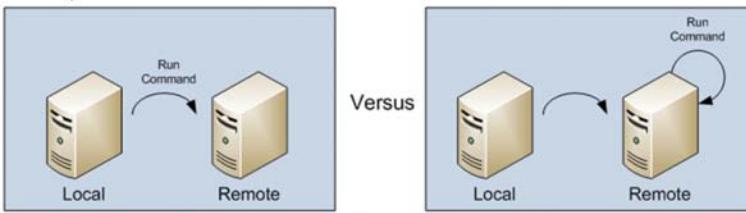
Logging onto the console of a server—either physically or by means of a Remote Desktop connection—is not a good management practice. Sometimes it's unavoidable, but having a server generate and maintain a graphical user environment, such as your desktop and the applications you run on it, is a poor use of server disk, memory, and processor resources.

Microsoft has gradually added remote management capabilities to many of the administrative tools and consoles that administrators use most, including Active Directory Users and Computers, Server Manager, and so forth. However, these are usually graphical tools, meaning they can be easy to use, but they can't be easily automated.

With Windows PowerShell's remote management capabilities, you can run nearly any Windows PowerShell command and have it execute on one or more remote computers. This capability enables you to manage remote computers more easily and with less overhead than a console session or a Remote Desktop connection.

## Many Forms of Remoting

- The actual process of remote administration with PowerShell can involve different approaches:
  - Cmdlets with a `-computerName` parameter don't use PowerShell remoting. They use RPC calls, which are not firewall friendly.
  - Some cmdlets are designed to work with a remote computer by default. These cmdlets use protocols related to their specific technology.
  - Windows PowerShell Remoting involves connecting to a remote computer, and then running a command locally on that computer.



### Key Points

The term *remote administration* means a lot of different things in Windows PowerShell.

- With the exception of some specific cmdlets you will learn about in this lesson, cmdlets that have a `-computerName` parameter do *not* use Windows PowerShell Remoting. Those cmdlets, such as `Get-WmiObject` or `Get-Service`, implement their own remote connectivity. `Get-WmiObject`, for example, uses the Remote Procedure Call (RPC) protocol that is intrinsic to the WMI technology.
- Some cmdlets are designed always to communicate with a remote computer. The Active Directory cmdlets, for example, know how to connect to a domain controller to do their work. Microsoft Exchange Server cmdlets know how to connect to an Exchange Server computer, even if the cmdlets are run on a Windows® 7 client computer. These cmdlets use communications protocols related to their specific technology and do not use Windows PowerShell Remoting.
- Windows PowerShell Remoting, which is what you will learn about in this lesson, is capable of making remote connections to one or more computers and running commands that are located *on those computers*. Windows PowerShell Remoting can be used with any cmdlet that is installed *on the remote computers* and does not rely on those cmdlets having a `-computerName` parameter or any other remote connectivity capability.

You can often combine these three techniques. For example, suppose you want to retrieve WMI information from a remote computer. You could use either of these techniques:

- Use `Get-WmiObject` and its `-computerName` parameter. The cmdlet runs on your local computer, and uses RPCs to connect to the WMI service on the remote computer. The remote service processes the WMI request and returns the results to your computer.
- Use Windows PowerShell Remoting to establish an HTTP connection to the remote computer. Using that connection, instruct the remote computer to run `Get-WmiObject`. That cmdlet runs on the remote computer and communicates with that computer's WMI service to complete the request. The WMI results are transmitted back to your computer over the HTTP connection.

You can see that there are some subtle differences between the techniques. A different protocol is used for communications, which may impact your ability to work through firewalls. Processing occurs in a different place in each situation. In the second example, most of the processing occurs on the remote computer, while in the first, the workload is split between the two computers.

The idea is that different forms of remote administration have uses in different scenarios. No one form is inherently better than another, although in specific situations, such as when you need to distribute processing or when you need to communicate through a firewall, one form of remoting may be more suitable.

Throughout the remainder of this lesson, you'll focus on Windows PowerShell Remoting.

MCT USE ONLY. STUDENT USE PROHIBITED

## Windows PowerShell Remoting

- The purpose of PowerShell Remoting is to...
  - Connect to a remote computer
  - Run one or more commands on that computer
  - Bring those results back to your computer
- The goals of PowerShell Remoting are...
  - Single seat administration
  - Batch administration
  - Lightweight administration
- The three ways to use PowerShell Remoting are...
  - 1-to-1 Remoting
  - 1-to-Many (or Fan-Out) Remoting
  - Many-to-1 (or Fan-In) Remoting



### Key Points

The purpose of Windows PowerShell Remoting is to connect to one or more remote computers, to run commands on those computers, and to bring the results back to your computer. This enables *single-seat administration*, or the ability to manage the computers on your network from your client computer, rather than having to physically visit each computer. A key goal of Windows PowerShell Remoting is to enable *batch administration*, which allows you to run commands on an entire set of remote computers simultaneously, something that older technologies like Remote Desktop cannot do.

Windows PowerShell Remoting (or simply *Remoting* for short) does not use the same connectivity technologies or protocols as older remote administration techniques such as Remote Desktop, and Remoting is generally lighter weight—meaning it involves less overhead—than some of those older technologies. This isn't to say that those older technologies aren't still useful and don't still have a place, but rather that Remoting can be more efficient and effective in some scenarios.

There are three main ways to use Remoting:

- **1-to-1 Remoting:** In this scenario, you connect to a single remote computer and run shell commands on it, exactly as if you had logged into the console and opened a Windows PowerShell window.
- **1-to-Many Remoting, or Fan-Out Remoting:** In this scenario, you issue a command that will be executed on one or more remote computers in parallel. You are not working with each remote computer interactively; rather, your commands are issued and executed in a batch and the results are returned to your computer for your use.
- **Many-to-1 Remoting, or Fan-In Remoting:** This scenario involves multiple administrators making remote connections to a single computer. Typically, those administrators will have differing permissions on the remote computer and might be working in a *restricted runspace* within the shell. This scenario usually requires custom development of the restricted runspace and will not be covered further in this course.

Remoting requires that Windows PowerShell v2, along with supporting technologies like Windows Remote Management (WinRM) and .NET Framework v2.0 or higher, be installed both on your computer and on any remote computers to which you want to connect. For security purposes, Remoting must be explicitly enabled before you can use it. You will learn how to enable Remoting later in this lesson.

MCT USE ONLY. STUDENT USE PROHIBITED

## WinRM

- PowerShell Remoting rides on top of the Windows Remote Management Service (WinRM).
  - WinRM is a Microsoft implementation of Web Services for Management (WS-MAN).
  - Unlike RPC, WinRM uses a single, definable port, making it easier to pass through firewalls.
  - WinRM can use TLS/SSL to encrypt data being passed through the network.
  - WinRM uses Active Directory's Kerberos for authentication.
  - WinRM is disabled by default on all computers. It can be enabled locally on a single computer or via Group Policy.
- WinRM v2 uses 5985/TCP as its default port for communication. This is different from WinRM v1, which used 80/TCP.

### Key Points

Rather than using older protocols such as RPCs to communicate, Remoting uses Windows Remote Management, or WinRM.

WinRM is a Microsoft implementation of Web Services for Management, or WS-MAN, a set of protocols that has been widely adopted across different operating systems. As the name implies, WS-MAN—and WinRM—uses Web-based protocols. An advantage to these protocols is that they use a single, definable port, making them easier to pass through firewalls than older protocols that randomly selected a port.

WinRM communicates via the Hypertext Transport Protocol, or HTTP. Note that the version of WinRM included with and used by Windows PowerShell v2 does not by default use ports 80 or 443, the traditional ports used for HTTP. By default, the current version (v2.0) of WinRM uses TCP port 5985 for incoming unencrypted connections. This port assignment is configurable, although leaving it at the default is recommended since Windows PowerShell's Remoting-related cmdlets assume the same default port.

**Note:** Older versions of WinRM used TCP port 80 (or 443 for encrypted connections) by default. That is not true with WinRM v2. However, WinRM v2 can be instructed to operate in *compatibility mode*, where it also listens on ports 80 and/or 443 to support older applications that only know to send WinRM requests to those ports. We will not cover that configuration in this course.

WinRM can use Transport Layer Security (TLS, sometimes inaccurately referred to as SSL) to encrypt the data being passed across the network. The individual applications that use WinRM, such as Windows PowerShell, can also apply their own encryption to the data that is passed to the WinRM service.

WinRM supports authentication and, by default, uses Active Directory's native Kerberos protocol in a domain environment. Kerberos does not pass credentials across the network and it supports mutual authentication to ensure that you are really connecting to the remote computer you specified.

For security purposes, WinRM is disabled by default and must be enabled. It can be enabled locally on a single computer or it can be enabled and configured by means of a Group Policy object (GPO).

MCT USE ONLY. STUDENT USE PROHIBITED

## How WinRM Works

- WinRM operates as a Windows service
  - Once enabled, that service listens on the assigned port for external requests
  - WinRM examines inbound requests to determine the application for which the request is meant
  - Applications must register an endpoint with WinRM, telling the service that the application is available and capable of processing WinRM requests
  - Kerberos (for authentication) and TLS/SSL (for encryption) are used during communication
- You must specifically register PowerShell as an endpoint to receive remoting requests, for security purposes

**Enable-PSRemoting**

### Key Points

WinRM runs as a service. When enabled, and provided the necessary firewall ports are opened, the WinRM service listens for incoming WinRM requests on its assigned port. When a request is received, WinRM examines the request to determine the application for which the request is meant.

Applications must register an *endpoint* with WinRM. Registration informs WinRM that the application is available and capable of processing incoming WinRM requests. This registration is what enables WinRM to *find* the application when an incoming request is received for that application.

When an incoming request is received, WinRM passes the request off to the designated application. The application does whatever it needs to do, and then passes any results back to WinRM. WinRM then transmits those results back to the computer and application that sent the original WinRM request.

Windows PowerShell does not, by default, register itself as an endpoint—again, this is for security purposes. You must explicitly permit the shell to register itself as an endpoint with WinRM. This can be done locally for a single computer or by means of a Group Policy object in a domain environment. You must also ensure that any necessary firewall configurations are modified to permit the WinRM traffic.

**Note:** You only need to enable WinRM, register an endpoint, and modify the firewall on computers that will *receive* WinRM requests. Any computer with Windows PowerShell v2 installed can *send* WinRM requests.

## Discussion: Enabling WinRM

- Are there any challenges to implementing WinRM in your environment?
  - Network challenges?
  - Security challenges?
  - Configuration control challenges?
  - Corporate policy or compliance challenges?



### Key Points

Are there any challenges to implementing WinRM in your environment?

Numerous remote management technologies are already being used in your environment, including RPC, Lightweight Directory Access Protocol (LDAP), and more. WinRM is simply another tool, and unlike some others it uses a single configurable port for its communications and supports a rich security configuration that enables you to configure it for exactly the communications you require.

## Remote Shell

- The **Enable-PSRemoting** cmdlet...
  - Enables WinRM
  - Starts the WinRM service
  - Sets the WinRM service to start automatically
  - Modifies the Windows Firewall configuration to permit incoming WinRM connections
  - Registers both the 64- and 32-bit versions of Windows PowerShell as WinRM endpoints as appropriate
- **Enable-PSRemoting** has an impact level of *high*, so you will be prompted to confirm your choice to allow the cmdlet to run
- Alternative: Group Policy!

### Key Points

There are two ways to configure Windows PowerShell and WinRM. The first, and perhaps easiest, is to run the **Enable-PSRemoting** cmdlet.

**Note:** Avoid running Set-WSManQuickConfig. This cmdlet does configure a portion of Remoting, but not all of it. In fact, Enable-PSRemoting runs Set-WSManQuickConfig internally and then performs the other actions needed to register Windows PowerShell as a WinRM endpoint.

You need to be logged on as an Administrator (and, if User Account Control is enabled, you must run the shell as Administrator) to successfully run Enable-PSRemoting. The cmdlet will:

- Enable WinRM.
- Start the WinRM service.
- Set the WinRM service to start automatically.
- Modify the Windows Firewall configuration to permit incoming WinRM connections.
- Register both the 64- and 32-bit versions of Windows PowerShell as WinRM endpoints.
- Have an impact level of *high*, so you will be prompted to confirm your choice to allow the cmdlet to run.

## Demonstration: Enabling Remote Shell

- Learn how to enable Remoting



### Key Points

In this demonstration, you will learn how to enable Remoting.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to all virtual machines with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Enabling Remote Shell.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## 1-to-1 Shell

- The 1-to-1 shell creates a remote command-line prompt inside your local PowerShell session
  - Similar to Telnet, SSH, or PSEExec \\computerName cmd
  - Running commands in the remote command-line prompt executes those commands on the remote system

**Enter-PSSession -comp server-r2**

- To exit the remote session, use...

**Exit-PSSession**

- Remember, all 1-to-1 shell commands execute **locally** on the **remote** computer

### Key Points

1-to-1 shell enables you to get a remote command-line prompt, similar to SSH or Telnet in a Unix operating system. 1-to-1 shell is also similar in function to the way PSEExec works.

**Note:** PSEExec is a Sysinternals utility used by many administrators to execute commands on remote computers. SSH and Telnet are remote connectivity tools often used in Unix or Linux environments. If you are not familiar with these, don't worry. They are presented here only as an analogy for those administrators who may have used them in the past.

Once you have opened a 1-to-1 remote shell connection, you are effectively using the remote computer's Windows PowerShell session as if you were physically sitting in front of the computer and had opened a Windows PowerShell window.

Use Enter-PSSession to establish the remote shell connection. This cmdlet accepts a –computerName parameter, which accepts the name or IP address of the computer you want to connect to. Other parameters enable you to specify alternate TCP ports, specify alternate user credentials, and so forth. Read the help for the cmdlet to learn about these and other capabilities and features.

Once the remote shell session is active, your Windows PowerShell prompt will change to indicate the name of the computer you are connected to, as well as the current path:

```
PS C:\> enter-pssession -comp server-r2  
[server-r2]: PS C:\Users\Administrator\Documents>
```

When you finish using the remote connection, run **Exit-PSSession** to return to your local shell prompt:

```
[server-r2]: PS C:\Users\Administrator\Documents> exit-pssession  
PS C:\>
```

When establishing a remote shell connection, remember that profile scripts (which are discussed in Module 8) are not executed, even if a profile script exists.

When you are using a remote shell connection, you only have access to the commands that are installed on that remote computer and that you have loaded into the shell by using Add-PSSnapin or Import-Module. The commands available on the remote computer may be different from those available on your local computer.

Use caution: When you are in a remote shell session, every command you run will be executed by the remote computer. Consider this sequence of commands:

```
Enter-PSSession -computerName Server1  
Enter-PSSession -computerName Server2  
Enter-PSSession -computerName Server3
```

The first command establishes a connection from your computer to Server1. The second occurs *within* that session, meaning you are connecting from Server1 to Server2. The third command occurs within *that* session, meaning you are connecting from Server2 to Server3. Try to avoid doing this, as each session involves a certain amount of memory, processing, and network overhead. Instead, exit a session before establishing a new one:

```
Enter-PSSession -computerName Server1  
Exit-PSSession  
Enter-PSSession -computerName Server2  
Exit-PSSession  
Enter-PSSession -computerName Server3  
Exit-PSSession
```

This sequence ensures that the connections to Server1, Server2, and Server3 are each made from your computer.

## Demonstration: Using 1-to-1 Remoting

- Learn to use 1-to-1 Remoting



### Key Points

In this demonstration, you will learn to use 1-to-1 Remoting.

### Demonstration steps:

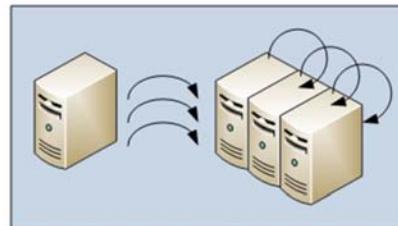
- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Using 1 to 1 Remoting.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## 1-to-Many Remoting

- 1-to-Many Remoting is useful when you need to execute the same command on multiple computers

- With 1-to-Many Remoting...

- You submit a set of commands to a list of computers
- Each computer executes the set of commands
- Then, each computer returns its result back to your computer



```
Invoke-Command -scriptblock { Dir c:\demo }  
-computerName Server1,Server2,Server3
```

- Invoke-Command runs synchronously, which means you must wait for it to finish before running more commands

### Key Points

1-to-1 Remoting is useful when you only need to run commands on a single remote computer. However, if you need to run the same commands on multiple remote computers, using 1-to-1 Remoting is inefficient. You have to enter a session on the first computer, run your command, and then exit. Then you enter a second session, run the command, and exit. You repeat this process for each remote computer, and it becomes overly time-consuming and repetitive.

1-to-Many Remoting enables you to submit a command, multiple commands, or even an entire script to a set of remote computers. The commands are transmitted to the remote computers by means of WinRM. Each computer executes the commands on its own and returns the results, again by means of WinRM, to your computer.

By default, the shell will not try to contact more than 32 computers at once. If you specify more than 32 computers, it will work with the first 32, and then, as those complete the commands, the shell will start contacting additional computers. This feature is called *throttling*, and you can modify the throttle level by means of a throttle parameter.

Using 1-to-Many Remoting is accomplished by means of the **Invoke-Command** cmdlet. The cmdlet accepts either a script block or the path to a script file and can accept one or more computer names:

```
Invoke-Command -scriptblock { Dir c:\demo }  
-computerName 'Server1', 'Server2'
```

This example would submit the **Dir c:\demo** command to both Server1 and Server2. Additional parameters of **Invoke-Command** enable you to specify an alternate TCP port, specify that encryption be used, specify alternate credentials, and so forth. Review the help for the cmdlet to learn more.

Note that **Invoke-Command** normally runs *synchronously*, meaning you have to wait until the shell finishes working with all remote computers before running additional commands. The shell displays the results as they come back from the remote computers. By specifying the **-AsJob** parameter, you can have

Invoke-Command runs *asynchronously*, as a background job. You can then run other commands immediately, and Invoke-Command executes in the background. Results from remote computers are stored as the results of the job.

MCT USE ONLY. STUDENT USE PROHIBITED

## Pipeline Binding

- Recall that some cmdlets bind their pipeline input ByPropertyName. This means that property names must match.
  - Problem: The –computerName parameter of Invoke-Command binds ByPropertyName, but some cmdlets produce objects that use the Name property instead.
  - Solution: Create custom objects to *change the property name*.

```
Get-ADComputer -filter * |  
Select  
@{Label='ComputerName';Expression={$.Name}} |  
Get-Service -name *
```

### Key Points

The –computerName parameter of Invoke-Command binds pipeline input ByPropertyName, meaning any input objects that have a ComputerName property will have that property's values attached to the –computerName parameter.

If you query computers from Active Directory, you notice that they have a Name property:

```
Import-Module ActiveDirectory  
Get-ADComputer -filter *
```

You can use Select-Object to create a custom object that has a ComputerName property, and then pipe those objects to Invoke-Command:

```
Get-ADComputer -filter * |  
Select @{Label='ComputerName';Expression={$.Name}} |  
Get-Service -name *
```

This example would return a list of all services on every computer in Active Directory.

## Working with Results

- Recall: PowerShell cmdlets return objects as their output.
- This poses a problem...
  - Software objects **cannot** be transmitted over the network.
  - To compensate, WinRM serializes those objects into XML for transfer, then deserializes them back into objects at your computer.
  - This changes how remote computer results data look, reverting them to text-based snapshots of those objects. It also strips away the methods associated with the object.
- To maintain some of the object-oriented flexibility, PowerShell adds a few additional properties to deserialized objects.
  - PSComputerName enables you to sort and group objects.

```
Invoke-Command -script { Get-Service } -computer  
Server1,Server2 | Sort PSComputerName | Format-  
Table -groupby PSComputerName
```

### Key Points

Keep in mind that Windows PowerShell cmdlets return objects as their output. Unfortunately, software objects cannot be transmitted across a network. Instead, WinRM *serializes* the objects into an XML format, since XML is just text, and text can be transmitted across a network quite easily. The objects are received by your computer and then *deserialized* back into objects. This conversion to and from XML does have an impact on how you can use the objects.

For example, consider the output of this command:

```
PS C:\> get-service | get-member
```

TypeName: System.ServiceProcess.ServiceController

Name	MemberType	Definition
---	-----	-----
Name	AliasProperty	Name = ServiceN
RequiredServices	AliasProperty	RequiredService
Disposed	Event	System.EventHan
Close	Method	System.Void Clo
Continue	Method	System.Void Con
CreateObjRef	Method	System.Runtime.
Dispose	Method	System.Void Dis
Equals	Method	bool Equals(Sys
ExecuteCommand	Method	System.Void Exe
GetHashCode	Method	int GetHashCode
GetLifetimeService	Method	System.Object G
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object I
Pause	Method	System.Void Pau
Refresh	Method	System.Void Ref
Start	Method	System.Void Sta
Stop	Method	System.Void Sto

MCT USE ONLY. STUDENT USE PROHIBITED

ToString	Method	string ToString
WaitForStatus	Method	System.Void Wai
CanPauseAndContinue	Property	System.Boolean
CanShutdown	Property	System.Boolean
CanStop	Property	System.Boolean
Container	Property	System.Componen
DependentServices	Property	System.ServiceP
DisplayName	Property	System.String D
MachineName	Property	System.String M
ServiceHandle	Property	System.Runtime.
ServiceName	Property	System.String S
ServicesDependedOn	Property	System.ServiceP
ServiceType	Property	System.ServiceP
Site	Property	System.Componen
Status	Property	System.ServiceP

This is a normal service object. Compare this with the service object returned from Invoke-Command:

```
PS C:\> invoke-command -script { get-service } -computer server-r2 | get-member
```

```
TypeName: Deserialized.System.ServiceProcess.ServiceController
```

Name	MemberType	Definition
ToString	Method	string ToString(), st
Name	NoteProperty	System.String Name=AD
PSComputerName	NoteProperty	System.String PSCompu
PSShowComputerName	NoteProperty	System.Boolean PSShow
RequiredServices	NoteProperty	Deserialized.System.S
RunspaceId	NoteProperty	System.Guid RunspaceI
CanPauseAndContinue	Property	System.Boolean {get;s
CanShutdown	Property	System.Boolean {get;s
CanStop	Property	System.Boolean {get;s
Container	Property	{get;set;}
DependentServices	Property	Deserialized.System.S
DisplayName	Property	System.String {get;se
MachineName	Property	System.String {get;se
ServiceHandle	Property	System.String {get;se
ServiceName	Property	System.String {get;se
ServicesDependedOn	Property	Deserialized.System.S
ServiceType	Property	System.String {get;se
Site	Property	{get;set;}
Status	Property	System.String {get;se

You can see that the type name now indicates that these objects are deserialized. In addition, the object's methods are no longer available (except a built-in ToString method that is attached to all objects); only the objects' properties have survived the conversion to and from XML. These objects are no longer attached to real, functioning pieces of software. Instead, they are essentially a text-based snapshot, or representation, of those objects.

Also note that a few additional properties have been attached to the objects' PSComputerName, for example, which contains the name of the computer that each object came from. When you are receiving results from multiple computers, this additional property can enable you to sort and group the objects by the computer they came from. For example:

```
Invoke-Command -script { Get-Service } -computer Server1,Server2 |
Sort PSComputerName | Format-Table -groupby PSComputerName
```

If you use the –AsJob parameter of Invoke-Command, then each computer you specify is submitted as a separate child job. So this command:

```
Invoke-Command -script { Get-Process } -computer Svr1,Svr2 -asjob
```

creates three jobs: a top-level job for `Invoke-Command` itself, and then a child job each for `Svr1` and `Svr2`. The results from each computer are stored in the corresponding child job. You can receive the results from those individually or you can receive all of the results from the top-level job.

MCT USE ONLY. STUDENT USE PROHIBITED

## Multi-Computer Jobs

- Jobs that span across multiple computers create top-level jobs, along with child jobs
  - Top-level jobs represent the multi-computer job as a whole
  - Child jobs represent the job on each remote computer
- To view the names of child jobs...  
**Get-Job -id 1 | Select childjobs**
- To retrieve child jobs...  
**Get-Job -name job2**
- To retrieve child job results...  
**Receive-Job -name Job2 -keep**

### Key Points

When you use Invoke-Command and its –AsJob parameter, you create a top-level job:

```
PS C:\> invoke-command -script { get-service }  
      -computer server-r2,localhost -asjob
```

Id	Name	State
--	----	-----
1	Job1	Running

You can check the status of this top-level job by running Get-Job, and once the job has completed, you can retrieve the results from all computers by using Receive-Job:

```
Receive-Job -id 1 -keep
```

However, it is also possible to see the names of the child jobs that have been created:

```
get-job -id 1 | select childjobs  
ChildJobs  
-----  
{Job2, Job3}
```

Once you know the names of the child jobs, you can retrieve them individually:

```
PS C:\> get-job -name job2
```

Id	Name	State
--	----	-----
2	Job2	Completed

You can also receive the results of each job individually if desired:

```
Receive-Job -name Job2 -keep
```

**Note:** The Invoke-Command cmdlet also has a `-JobName` parameter, which enables you to specify a custom name for the job rather than using automatically-generated job names like `Job2`.

MCT USE ONLY. STUDENT USE PROHIBITED

## Keep Workloads Remote

- As a best practice, try to execute as much processing on remote systems as possible
  - This practice is computationally less expensive
- Example: Avoid this...

```
Invoke-Command -script { Get-Service } -computer Server1 |  
Where { $_.Status -eq "Running" } |  
Sort Name |  
Export-Csv c:\services.csv
```

- Do this instead...

```
Invoke-Command -script {  
Get-Service |  
Where { $_.Status -eq "Running" } |  
Sort Name  
} | Export-Csv c:\services.csv
```

### Key Points

As a best practice, try to do *all* of your processing on the remote computer. The results you receive from Invoke-Command should be in the final form that you want, although you may choose to format them, export them to a file, or direct them to a different output device on your local computer.

For example, avoid the following, which retrieves *all* of the services from a remote computer and then performs sorting and filtering locally:

```
Invoke-Command -script { Get-Service } -computer Server1 |  
Where { $_.Status -eq "Running" } |  
Sort Name |  
Export-Csv c:\services.csv
```

Instead, do as much of the work as possible on the remote computer:

```
Invoke-Command -script {  
Get-Service |  
Where { $_.Status -eq "Running" } |  
Sort Name  
} | Export-Csv c:\services.csv
```

When submitting a command to multiple computers, having each computer perform as much of their processing as possible helps to better distribute the overall workload of your task.

## Demonstration: Using 1-to-Many Remoting

- Learn to use 1-to-Many Remoting



### Key Points

In this demonstration, you will learn to use 1-to-Many Remoting.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Using 1 to Many Remoting.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Restricted Runspaces

- Recall: Restricted runspaces are created by developers to limit administrative capabilities on that server
  - A hosted Exchange provider may limit your PowerShell Remoting capabilities to only certain portions or areas of the Exchange configuration
  - PowerShell restricted runspaces provide this capability
- Restricted runspaces can...
  - Remove certain parameters from a cmdlet, or replace parameters with fixed values
  - Add new cmdlets, remove cmdlets
  - Add, remove, or change parameters
- Be aware that some cmdlets may not behave exactly as expected if restricted runspaces have been implemented

### Key Points

Earlier, you learned about the concept of restricted runspaces, which applies to the Many-to-1 Remoting scenario.

A restricted runspace is usually created by a software developer and deployed on a server that runs Windows PowerShell. The intent of a restricted runspace is to give remote administrators limited administrative capabilities on that server.

For example, in a hosted Exchange Server environment, you might share space on an Exchange Server computer with other customers. Being able to use Windows PowerShell Remoting to control your portions of the Exchange configuration, such as your users' mailboxes, is useful, but the hosting company must be able to restrict you from modifying the configuration of other customers' portions of the server.

A restricted runspace provides this capability. The runspace might remove certain parameters from a cmdlet, for example, and replace them with fixed values queried from a database. This might, for example, allow you to create new mailboxes on an Exchange Server, but limit you to creating them on a predetermined server queried from the hosting company's customer database.

Restricted runspaces can add new cmdlets, remove cmdlets, add, remove, or change parameters, and more. As an administrator using a restricted runspace, you cannot bypass these restrictions; you just need to be aware that some cmdlets may not behave exactly the way you are used to, or as you have read about.

## Disadvantages of Specifying Connection Details

- Having to specify connection details over and over for many different connections is repetitive and can cause errors
- PowerShell offers a way to create and reuse connections
- The New-PSSession cmdlet...
  - Uses parameters that are similar to Enter-PSSession and Invoke-Command
  - Creates a persistent session object, which can be stored in a variable for later use
  - Maintains this persistent session object for as long as the shell window remains open

```
$sessions = New-PSSession -computer Srv1,Srv2,Srv3  
Invoke-Command -script { Get-Service } -session  
$sessions
```

```
$sessions = New-PSSession -computer Srv1,Srv2,Srv3  
Enter-PSSession -session $sessions[1]
```

### Key Points

So far, you have specified connection information for both 1-to-1 and 1-to-Many Remoting. That is, when running either Invoke-Command or Enter-PSSession, you have provided computer names and possibly provided port numbers, alternate credentials, authentication options, and so forth. Having to do this repeatedly can cause errors, and so Windows PowerShell offers a way for you to create and re-use connections. The shell provides this capability through a *PSSession*.

You create a new, persistent connection by means of the New-PSSession cmdlet. Its parameters are similar to those of Enter-PSSession and Invoke-Command: You specify computer names, credentials, ports, and so on. The result is a persistent session object, which can be stored in a variable for later use. The shell maintains the session for as long as the shell window is open. Closing the shell automatically closes the connections.

Session objects can be passed to both Invoke-Command and Enter-PSSession, although Enter-PSSession only accepts a single session object. For example:

```
$sessions = New-PSSession -computer Srv1,Srv2,Srv3  
Invoke-Command -script { Get-Service } -session $sessions
```

This will invoke Get-Service on all three computers. Or:

```
$sessions = New-PSSession -computer Srv1,Srv2,Srv3  
Enter-PSSession -session $sessions[1]
```

This will enter a 1-to-1 Remoting session on Srv2, the second session object contained in the \$sessions variable.

## Best Practices

- Some best practices...
  - Always use Start-Job to start a local background job
  - Never use Invoke-Command and -AsJob to start a local background job
- Using Invoke-Command to start a local job creates a loopback situation
  - The local WinRM service communicates with **itself** over the network
  - This creates some tricky security situations

## Key Points

Always use Start-Job to start a local background job—never use Invoke-Command and its -AsJob parameter. Start-Job ensures that the job is created locally, and that WinRM isn't used.

If you do use Invoke-Command to start a local job, be aware that you are engaging in a *loopback*. That is, WinRM goes out to the network, and then communicates with itself over the network. This can create tricky security situations. For example, this command:

```
Invoke-Command -script { Get-EventLog Security } -comp localhost
```

actually involves network communications, and can place some restrictions on what the submitted commands are able to do.

## Session Management

- Sessions require on-system resources to remain open
  - They consume memory, processor, and networking overhead
  - It's a good idea to close sessions when you're finished with them
- The Remove-PSSession cmdlet...
  - Closes sessions that are no longer necessary
  - Can use variables to close individual sessions or sets of sessions

```
$sessions | Remove-PSSession
```

```
Get-PSSession | Remove-PSSession
```

### Key Points

Sessions do require some memory, processor, and networking overhead, so you should not leave them open without need. Use the Remove-PSSession cmdlet to close sessions and use Get-PSSession to retrieve sessions that are currently opened.

Closing all of the sessions that you have stored in a variable is easy. Use this command:

```
$sessions | Remove-PSSession
```

You could easily create and maintain several sets of sessions in different variables. To close every open session, use this command:

```
Get-PSSession | Remove-PSSession
```

## Demonstration: Session Management

- Learn to manage sessions



### Key Points

In this demonstration, you will learn to manage sessions.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod07\Democode\Session Management.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Implicit Remoting

- The topic of implicit remoting was first introduced in Module 5.
  - Recall: Some cmdlets will not run on some operating systems. For example, the ActiveDirectory cmdlet will not run on Windows XP. Implicit remoting enables cmdlets from a more-capable computer to be used on less-capable computer.
- Three basic steps:
  - Use New-PSSession to establish a session from the Windows XP computer to the domain controller that contains the ActiveDirectory module.
  - Invoke a command on the domain controller to load the ActiveDirectory module.
  - Use the **Import-PSSession** cmdlet to import the remote commands to the Windows XP computer. Specify that only commands whose nouns start with *AD* be exported, which ensures that only the Active Directory cmdlets are included in the export.

### Key Points

A final application of Remoting is called *implicit remoting*. The idea is to recognize that different computers in your environment contain different commands. For example, a domain controller may contain the ActiveDirectory module. That module will not run on a Windows XP client computer, although Windows XP can have Windows PowerShell v2 installed. Suppose you have a Windows XP client computer and you want to use the cmdlets in the ActiveDirectory module. Implicit remoting provides a way to do so.

The basic steps involved are as follows:

1. Use New-PSSession to establish a session from the Windows XP computer to the domain controller that contains the ActiveDirectory module.
2. Invoke a command on the domain controller to load the ActiveDirectory module.
3. Use the Import-PSSession cmdlet to import the remote commands to the Windows XP computer. Specify that only commands whose nouns start with *AD* be exported, ensuring that only the Active Directory cmdlets are included in the export. You can also specify a prefix for the remote commands' noun, to remind you that they're remote.

This process does *not* install the ActiveDirectory module on the Windows XP computer—that module can't run on Windows XP. To use the Active Directory cmdlets, you need to run them remotely. You can execute the remote cmdlets by specifying their normal cmdlet name, plus the noun prefix you specified.

Of course, implicit remoting works in *any* scenario where you want to use the cmdlets installed on another computer—you aren't limited to the ActiveDirectory module or to Windows XP, although that is a good example of how implicit remoting can be useful.

Note that the implicit commands are only available while that original session is active. If you close the shell or that session, you lose the remote cmdlets.

## Demonstration: Implicit Remoting

- Learn to use implicit remoting



### Key Points

In this demonstration, you will learn to use implicit remoting.

### Demonstration steps:

- 1 Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** and log on to virtual machine **LON-SVR1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-SVR1** at **E:\Mod07\Democode\Implicit Remoting.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Using Windows PowerShell Remoting

- Exercise 1: Interactive Remoting
- Exercise 2: Fan-Out Remoting
- Exercise 3: Fan-Out Remoting Using Background Jobs
- Exercise 4: Saving Information from Background Jobs

### Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 45 minutes**

### Estimated time: 45 minutes

You are a system administrator and often need to run commands on remote computers. You could ask each user to run the commands themselves, but you prefer to have control to run the commands yourself. You have heard about being able to run remote commands using WMI, and know that your company also has a desktop management system that may be able to run these commands, but you would rather be in complete control of the commands yourself. Furthermore, your company uses host firewalls and you know that they can cause issues with using WMI. So you've decided to implement PowerShell remoting throughout your organization.

### Lab Setup

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

- 1 Start the **LON-DC1**, **LON-SVR1**, and **LON-SVR2** virtual machines, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 2 Open a Windows PowerShell session as **Administrator** on all three virtual machines.
- 3 Run the following command on all three virtual machines:
  - **Enable-PSRemoting -Force**

## Exercise 1: Interactive Remoting

### Scenario

You often have to work with remote computers and want to open up an interactive remoting session to run commands interactively on other computers. The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Initiate a 1:1 remoting session to a remote computer.
2. Run remote commands interactively.

► **Task 1: Initiate a 1:1 remoting session to a remote computer**

- On LON-SVR1, initiate an interactive remoting session to LON-DC1.

► **Task 2: Run remote commands interactively**

- On LON-SVR1, using the interactive remoting session to LON-DC1, issue the command **Get-HotFix**.

**Results:** After this exercise, you will have initiated a 1:1 remoting session with LON-DC1 from LON-SVR1 and will have issued Get-HotFix remotely.

## Exercise 2: Fan-Out Remoting

### Scenario

You often have to check the event logs of remote servers to check for certain events and would like to use PowerShell remoting to ease the process of connecting to remote computers. The main tasks for this exercise are as follows:

1. Invoke a command on multiple computers.
2. Sort and filter objects in the pipeline.

► **Task 1: Invoke a command on multiple computers**

- From LON-SVR1, invoke the command **Get-EventLog -LogName Application -Newest 100** on LON-SVR2 and LON-DC1.

► **Task 2: Sort and filter objects in the pipeline**

- Run the same command again and group the results by the **EventId** property, sort them in descending order by the resulting **Count** property, then select only the first five to display.

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 from LON-SVR1 to retrieve a listing of the 100 newest events from the Windows Application event log, and will have grouped and sorted the results to display the top five EventId values that have occurred the most.

## Exercise 3: Fan-Out Remoting Using Background Jobs

### Scenario

You have an upcoming corporate audit and have to check the system drive of remote servers to check for certain permissions. You would like to use PowerShell remoting to ease the process of connecting to these remote computers. By experience, the auditing process can take a long time per server, so you will also want to start this as a background job so you can continue using your PowerShell console while the job is running. The main tasks for this exercise are as follows:

1. Invoke a command on multiple computers as a background job.
2. Wait for the parent job to complete.
3. Get the ID of all child jobs.
4. Sort and filter objects in the pipeline.

- ▶ **Task 1: Invoke a command on multiple computers as a background job**
  - From LON-SVR1, invoke the command “**Get-Acl -Path C:\Windows\System32\\* -Filter \*.dll**” on LON-SVR2 and LON-DC1 as a background job.
- ▶ **Task 2: Wait for the parent job to complete**
  - From LON-SVR1, invoke the cmdlet to wait until the previous background job completes.
- ▶ **Task 3: Get the ID of all child jobs**
  - From LON-SVR1, get a listing of all the IDs of the child jobs from the previous background job.
- ▶ **Task 4: Sort and filter objects in the pipeline**
  - Once the job is completed, retrieve the results and filter the output to only display files where the owner name contains *Administrator*.

**Hint:** Remember, you can use the -like operator and a comparison value such as \*Administrator\* to search for owner names that contain *Administrator*.

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 as a background job from LON-SVR1 to retrieve the ACLs on all the files in C:\Windows\System32 with the extension \*.dll, and will have filtered to display only the files with the owner being the *BUILTIN\Administrators*.

## Exercise 4: Saving Information from Background Jobs

### Scenario

For that same corporate audit, you need to verify information about % Processor Time PerfMon counters for multiple machines. You need to save information about one computer to a text file. The main tasks for this exercise are as follows:

- 1 Invoke a command on multiple computers as a background job.
- 2 Wait for the parent job to complete.
- 3 Receive the results only from a single child job.
- 4 Export the results to a file.

- ▶ **Task 1: Invoke a command on multiple computers as a background job**
  - From LON-SVR1, invoke the command **Get-Counter -Counter "\Processor(\_Total)\% Processor Time" -SampleInterval 2 -MaxSamples 3** on LON-SVR2 and LON-DC1 as a background job.
- ▶ **Task 2: Wait for the parent job to complete**
  - From LON-SVR1, invoke the cmdlet to wait until the previous background job completes.
- ▶ **Task 3: Receive the results only from a single child job**
  - From LON-SVR1, get the results of the child job that retrieved the results from LON-DC1.
- ▶ **Task 4: Export the results to a file**
  - From LON-SVR1, use the previous results of the child job that retrieved the results from LON-DC1 and save them to a file.

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 as a background job from LON-SVR1 to retrieve the total %Processor Time value, waited for the parent job to complete, and will have retrieved the results of the child job that was used for LON-DC1 only by saving the results to a file.

## Lab Review

1. Is there any limit to the number of remote sessions that can be run?
2. If I use Invoke-Command from one console and start it as a background job, can other consoles on the same server also see the background jobs and access their information directly?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- What is the primary difference between Start-Job and Invoke-Command (used with its –AsJob parameter)?
- Will implicit remoting work if the remote computer does not have WinRM enabled?
- What command would disable Remoting on the local computer?



### Review Questions

1. What is the primary difference between Start-Job and Invoke-Command (used with its –AsJob parameter)?
2. Will implicit remoting work if the remote computer does not have WinRM enabled?
3. What command would disable Remoting on the local computer?

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 8

## Advanced Windows PowerShell® Tips and Tricks

### Contents:

<b>Lesson 1:</b> Using Profiles	8-3
<b>Lesson 2:</b> Reusing Scripts and Functions	8-11
<b>Lesson 3:</b> Writing Comment-Based Help	8-24
<b>Lab:</b> Advanced PowerShell Tips and Tricks	8-30

## Module Overview

- Create and safely use a Windows PowerShell profile script
- Repackage scripts as script modules for easier distribution
- Add comment-based help to functions, scripts, and script modules



Any technology is easier to use when you know some of the "tips and tricks" for doing so more effectively. In this module, you will learn some of the tips and tricks that more advanced users take advantage of to make Windows PowerShell® more effective.

None of the topics covered in this module are critical to using Windows PowerShell—but they do make using it easier and more effective in many ways.

### Module Objectives

After completing this module, you will be able to:

- Create and safely use a Windows PowerShell profile script.
- Repackage scripts as script modules for easier distribution.
- Add comment-based help to functions, scripts, and script modules.

## Lesson 1

# Using Profiles

- Explain how profiles are executed and identify predetermined profile filenames and locations
- Explain the security concerns associated with profile scripts
- Create a profile script



Throughout this course, you have encountered many things that you can do with the shell that do not persist beyond the end of the shell session: importing formatting views, adding aliases, adding PSDrives, and so forth.

A profile enables you to create a shell environment that is reinitialized each time you open a new shell session.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain how profiles are executed and identify predetermined profile filenames and locations.
- Explain the security concerns associated with profile scripts.
- Create a profile script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Why Profiles?

- Each time you start a new PowerShell session, you begin with a fresh environment
  - Default configurations
  - Default aliases
  - Default everything!
- Profiles in Windows PowerShell provide a mechanism to customize the shell environment automatically with each new session
  - Preload modules, custom aliases, connect PSDrives, anything that is invoked via a valid PowerShell command
  - PowerShell searches for a predetermined file in a predetermined location for this script
  - You can change this script to customize your working environment

### Key Points

Each time you open a new Windows PowerShell session (technically called a *runspace*), you start with a fresh environment: all the default configurations, all the default aliases, and so on.

As you work with the shell, you load modules, create aliases, define functions, add PSDrives, and so forth. However, when you close that shell session, all of those customizations are lost—you start from scratch when you open a new session.

You might have elements of the shell that you want available each time you start a new session. Particular modules, custom aliases, PSDrives, and so forth may all be things that you use daily and want to use without having to manually add them each time.

A profile script gives you the ability to have that automatic, custom shell environment. It is a normal Windows PowerShell script, given a special predetermined name and kept in a special predetermined location. The shell looks for that filename in that location each time it starts, and it executes whatever commands are in the file. Those commands can include any valid Windows PowerShell command, including Import-Module, New-Alias, New-PSDrive, Add-PSSnapin, and so on.

## Discussion: What's in Your Profile?

- What kinds of things would you add to a Windows PowerShell profile script?



### Key Points

What kind of things would you add to a Windows PowerShell profile script?

## Profiles Do Not Always Run

- Profile scripts will not execute in certain circumstances:
  - When you connect to a remote instance of the shell, the profile script in the remote instance will not run
  - When the shell is embedded inside another application, such as the Exchange Server Manager application
  - When the shell's execution policy is set to Restricted, or if the execution policy is set to AllSigned and the profile script is not signed
- **Rule of thumb:** You should expect profile scripts to execute only when you are using an interactive shell window, via the console or Windows PowerShell ISE

### Key Points

There are a couple of instances where the shell does not execute a profile script, even if one exists.

When you connect to a remote instance of the shell, using remoting, that remote instance will not execute a profile script—even if one exists in the correct location on the remote computer.

When the shell is embedded inside another application—such as the Microsoft Exchange Management Console application—the shell does not execute a profile script. However, that other application is free to load and execute whatever scripts it wants to, and it may choose to execute the standard shell profile scripts if its developers programmed it to do so.

In general, then, you should expect profile scripts to execute only when you are using a shell window interactively, either using the console window or the Windows PowerShell ISE.

Also, of course, a profile script cannot execute if the shell's execution policy is set to Restricted or if the execution policy is set to AllSigned and the profile script is not signed.

## Profile Locations

- Profile scripts can be found
  - For the current user in \$Home\[My ]Documents\profile.ps1
  - For all users in \$PsHome\Profile.ps1
- Host applications can also load their own profile scripts
  - Current user for the console in \$Home\[My ]Documents\WindowsPowerShell\profile.ps1
  - All users for the console in \$PsHome\Microsoft.PowerShell\_Profile.ps1
  - Current user for the ISE in \$Home\[My ]Documents\WindowsPowerShell\Microsoft.PowerShellISE\_profile.ps1
  - All users for the ISE in \$PsHome\Microsoft.PowerShellISE\_profile.ps1

**Multiple profiles mean that each host will load up to four different profiles. To reduce complexity, consider standardizing on Current User profiles.**

### Key Points

There is no default Windows PowerShell profile, and the locations that the shell examines to find profile scripts do not necessarily exist by default.

Read the help for `about_profile` to read more about profiles and the correct locations and filenames for them. In general, the shell looks for the following:

- For the current user, for all Windows PowerShell host applications, the shell loads \$Home\[My ]Documents\profile.ps1.
- For all users, for all Windows PowerShell host applications, the shell loads \$PsHome\Profile.ps1

**Note:** \$Home and \$PsHome are predefined constants within the shell. You can type these constants at a command line and press Return to see what paths they contain. The "[My ]Documents" path indicates the user's Documents folder. On Windows Vista and newer versions of Windows, the path is \$Home\Documents; on Windows XP and Windows Server 2003, the path is \$Home\My Documents.

The concept of a *host application* adds additional possibilities for profile scripts. For example, the console window shell also looks for these profile scripts:

- Current user: \$Home\[My ]Documents\WindowsPowerShell\Profile.ps1
- All users: \$PsHome\Microsoft.PowerShell\_Profile.ps1

The Windows PowerShell ISE is a separate host application, meaning it hosts the shell engine and can define its own profile locations:

- Current user: \$Home\[My]Documents\WindowsPowerShell\Microsoft.PowerShellISE\_profile.ps1
- All users: \$PsHome\Microsoft.PowerShellISE\_profile.ps1

As a best practice, you should use only the *Current user* profiles. Keep in mind that neither the full folder path nor the file exists by default; you have to create them manually.

The reason there is an *all hosts* and a *per-host* profile is simple: you might want some things available no matter what kind of shell host you are using, such as a particular module or some custom aliases. There may, however, be things that you use only within the Windows PowerShell ISE, and so you can put those into that host-specific profile.

Most Windows PowerShell hosts will, then, load up to four profiles:

- The current-user, current-host profile
- The current-user, all-hosts profile
- The all-user, current-host profile
- The all-user, all-hosts profile

The profiles are loaded in that exact order.

You can also manually launch Windows PowerShell and use the `-Noprofile` command-line parameter to suspend profile execution.

## Profile Security

- **Caution:** Your profile script could be used as a vector for malware attack, because
  - It is a simple text file
  - It lives in your Documents folder
  - It executes automatically
- The best protection against this potential form of attack is to run anti-malware software on your computer
  - Also, do not store profile scripts on computers where you do not normally use the interactive shell or run scripts
  - On computers where you do not routinely run scripts, leave the shell execution policy at Restricted

### Key Points

Because the Windows PowerShell profile is a simple text file, lives in your Documents folder (in the case of the current user profiles, at least), and executes automatically, the profile can become a vector for malware. Even accidental changes can cause problems.

The scenario is this: You either have a profile script or not. A piece of malware either modifies your existing profile, or it creates one if you have not done so. The malware inserts malicious commands, such as removing Active Directory objects. The next time you open the shell—typically running it as an administrator—the profile, including the newly-inserted commands, executes automatically.

The best protection against this potential form of attack is to continue to run anti-malware software on your computer and to keep malware off of your computer entirely. Once your computer is compromised by malware, that malware can use many different means of misusing your administrator privileges and causing problems in your environment.

There is little point in having a profile on a computer where you do not normally use the shell interactively and where you also routinely run scripts. For computers on which you do not routinely run scripts, leave the shell execution policy at Restricted—that setting prevents profile scripts from running at all.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Profiles

- Learn how to use Windows PowerShell profile scripts



### Key Points

In this demonstration, you will learn to use Windows PowerShell profile scripts.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod08\Democode\Using Profiles.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 2

# Reusing Scripts and Functions

- Dot source a script into the shell or another script
- Package a script as a script module and import it into the shell
- Modify a script module to expose only selected functions when the module is imported into the shell



As you become more comfortable using Windows PowerShell, you may begin to write your own functions and scripts that accomplish various administrative tasks.

Being able to reuse scripts and functions—especially functions—in a variety of situations can save you work. One easy way to reuse a function is to simply copy and paste it into whatever script you need it. That approach, however, creates long-term maintenance issues because you have to change multiple copies of the function each time a change is needed.

The shell offers better options for reusing scripts and functions.

### Lesson Objectives

After completing this lesson, you will be able to:

- Dot source a script into the shell or another script.
- Package a script as a script module and import it into the shell.
- Modify a script module to expose only selected functions when the module is imported into the shell.

MCT USE ONLY. STUDENT USE PROHIBITED

## Functions

- A **function** is a way of writing your own command
  - In fact, some functions behave almost exactly like the cmdlets you've already been using
  - The last module in this course teaches how to create your own functions
  - But you may receive functions from colleagues, or you may find functions on the Internet that you want to use
- One way to re-use functions is to paste them into the script file where you want to use them
  - However, this is not necessarily efficient. If you later need to change that function, you'll need to change it everywhere.
  - Also, pasting a function into a script doesn't necessarily make that function available at the command line

This lesson will discuss a set of tactics you can use to enhance the re-use of your scripts

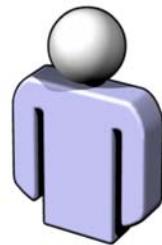
### Key Points

A *function* is a way of writing your own command. In fact, some functions behave almost exactly like the cmdlets you have already been using. The last module in this course teaches how to create your own functions, but you also may receive functions from colleagues or you may find functions on the Internet that you want to use.

One way to reuse functions is simply to paste them into whatever script file you want to use the function within. That is not very efficient, however, because if you decide to change the function someday, you will have to change it in every script where you pasted it. Also, pasting a function into a script does not make the function available directly from the command line—and you might obtain or create functions that you *want* to run, such as cmdlets, directly from the command line.

## Discussion: Script Reuse

- What kinds of functions or scripts do you think you might create that may be used in many different places?
  - Functions that check for computer connectivity
  - Functions that retrieve specific pieces of information from a computer
  - Functions that import or export data, such as from or to a database



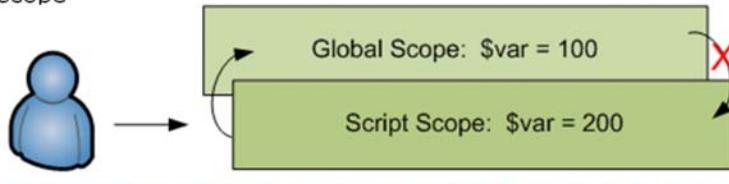
### Key Points

What kinds of functions or scripts do you think you might create that may be used in many different places?

- Functions that check for computer connectivity
- Functions that retrieve specific pieces of information from a computer
- Functions that import or export data, such as from or to a database

## Quick Concept: Scope

- **Recall:** PowerShell works with a complex hierarchy of scopes
- A scope acts as a sort of box, inside which various shell elements execute
  - A child scope can READ elements from its parents
  - A parent scope can never read or write anything in its child scopes
  - You can WRITE only to the current scope
  - If you try to change something in a parent scope, the result will be a new item of that same name being created in the current scope



### Key Points

The shell has a complex hierarchy of *scopes*. A scope acts as a sort of box inside which various shell elements execute. Almost everything an element does or creates exists within that box and when the element finishes, the box disappears, along with everything inside of it.

Scripts and functions are the most common examples of elements that execute inside a box. When a script runs, much of what it does and defines disappears when the script finishes running. This work includes adding PSDrives, defining aliases, defining variables, defining functions, and so forth.

The top-level scope is the shell itself, the box in which all other boxes are created. If something is created within the shell's top-level scope, that something will be available to everything else in the shell.

The practical result of this is that scripts, for example, are entirely self-contained. Suppose you paste a function into the script. You then go to the command line and run that script. The function becomes defined in memory when the script executes, but when the script completes, the function is removed from memory. That behavior is perfect if you only wanted to use that function from within that script. However, sometimes you may want a way to make the function *persistent*, so that it stays in memory after the script finishes executing. Doing so would enable you to use the function directly from the command line.

## Dot Sourcing

- Dot sourcing is a way of **eliminating the box** around a script. It is a means of including one script within another.
  - When you dot source a script, you are running it in your current scope rather than creating a new scope
  - Anything defined by the dot sourced script remains after the script finishes running, including variables, PSDrives, aliases, functions, and so forth
- Dot sourcing enables you to keep your reusable functions in one or more *library* scripts
  - When you need to use a library function, dot source the library script that contains the function
  - This process defines the function in the current scope, making it available in that scope for future use

.../library.ps1

### Key Points

*Dot sourcing* is a way of eliminating the box around a script. When you dot source a script, you are running it *in the current scope*, rather than creating a new scope for it to run within.

The result is that anything defined by the dot-sourced script remains after the script finishes running, including variables, PSDrives, aliases, functions, and so forth.

The practical use of dot sourcing is to keep your reusable functions in one or more *library* scripts. When you need to use one of those functions, dot source the library script that contains the function. Doing so defines the function inside the current scope, making the function available within that scope from that point on.

**Note:** In a way, you can think of dot sourcing as a means of *including* one script within another or *including* a script's contents in the shell itself.

You can dot source a script at the shell command prompt, and doing so retains everything that script defines at the global shell level. You can also dot source a script within another script. For example:

1. You run a script named ScriptA.ps1.
2. Inside ScriptA.ps1, you define a function named Function1.
3. Inside ScriptA.ps1, you dot source ScriptB.ps1.
4. Inside ScriptB.ps1, you define a function named Function2.
5. Inside ScriptA.ps1, you can now access both Function1 and Function2.
6. ScriptA.ps1 finishes running.
7. At the shell's command prompt, you cannot access Function1 or Function2. They were defined within the scope that was created for ScriptA.ps1; now that ScriptA.ps1 has completed, that scope and

everything inside of it is gone. Dot sourcing ScriptB.ps1 made its contents available only within the scope created for ScriptA.ps1; it did not elevate ScriptB.ps1 to the global scope.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Dot Sourcing

- Learn how to dot source a script



### Key Points

In this demonstration, you will learn to dot source a script.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod08\Democode\Using Dot Sourcing.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Script Modules

- Dot sourcing is effective, but it is also confusing
- Script modules offer a more formal and structured way to re-use scripts and functions
  - Easier to distribute to others
  - Offer the ability to include multiple files, such as views
- A script module is a normal shell script with a .psm1 extension that is stored in a predefined location
  - Predefined locations can be viewed with the command:

**Dir env:\psmodulepath**

- Modules are created and uniquely named in the script file. They are then subsequently loaded for use with the command:

**Import-Module ModuleName**

### Key Points

Dot sourcing is effective, but it can also be confusing, and it is a somewhat informal way of reusing scripts and functions.

Script modules offer a more formal, structured way of reusing scripts and functions. Script modules are also somewhat easier to distribute to colleagues, and when you use scripts, you can include multiple files, such as formatting views, along with the script module itself.

At their simplest, a script module is a normal shell script that has a .psm1 filename extension. There is no special formatting or structure required within the script to make it a module. You should, however, store the module in one of the predefined locations where the shell searches for modules. To see these locations, run this command:

**Dir env:\psmodulepath**

Typically, you select a name for your module—for example, MyModule. You create a folder with the module name in one of the predefined module path locations, such as \Documents\WindowsPowerShell\Modules\MyModule. You then store your module file in that folder, using a filename that consists of the module name and the .psm1 filename extension: MyModule.psm1, for example.

You can then load the module by using the Import-Module cmdlet:

**Import-Module MyModule**

When you import a module, any aliases, functions, PSDrives, or variables defined within the module become available in the global scope—meaning they become available throughout the shell. They remain in the shell until you either close the shell window or remove the module by using the Remove-Module cmdlet.

Note that the shell looks for filenames in a specific order when importing modules. For example, when you run Import-Module MyModule, the shell looks along the paths defined in the PSModulePath environment variable for a subfolder named MyModule. Within that, it first looks for a MyModule.psd1 file, which is a module manifest (which you will learn about in a moment). If that file is not found, it looks for MyModule.psm1, which is a stand-alone script module.

**Note:** To share modules located in a central location, such as on a file server, add the central location's path to the PSModulePath environment variable.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Script Modules

- Learn how to use Windows PowerShell script modules



### Key Points

In this demonstration, you will learn to use Windows PowerShell script modules.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod08\Democode\Using Script Modules.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## More Script Module Features

- Script modules can also include *internal*, or private functions
  - These functions are for use within the script itself, but are not exposed to users who import the module into their shell.
- The Export-ModuleMember cmdlet
  - Controls which items are exposed when the module is exported. Anything not explicitly exported will not be exposed, but will be available internally.
- Script modules can also contain manifests
  - Manifests list your script module and its supporting files, such as formatting views, that must be loaded with your module.
- The New-ModuleManifest cmdlet
  - Creates new manifest files, which are then customized to list the necessary files.

### Key Points

Script modules can do more than just provide an easier way of dot sourcing a script. For example, you may create a script module that includes *internal* or *private* functions. Your intent with these is that you need to use them from within other functions in the script module, but you do not want them exposed to users who import the module into their shell.

For ease of use, the Import-Module cmdlet normally exposes everything within the script module. However, by using the Export-ModuleMember cmdlet within your script module, you can control the items that are exposed when the module is imported. Anything not explicitly exported is not exposed, but is still available internally within the module itself.

You can also create a manifest for your module. Manifests are XML files with a filename that consists of the module name and a .psd1 filename extension. The manifest lists not only your script module, but can also list other supporting files—including formatting views—that must be loaded along with your module. Create a new manifest file by using the New-ModuleManifest cmdlet, and then customize the manifest to list the necessary files. Read the help for that cmdlet to learn more.

**Note:** When a module folder contains both a .psd1 file and a .psm1 file, the shell loads the .psd1 file. That, in turn, references the .psm1 file, causing the shell to load it next.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Export-ModuleMember

- Learn how to create private functions within a script module



### Key Points

In this demonstration, you will learn to create private functions within a script module.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod08\Democode\Export-ModuleMember.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Discussion: Sharing Modules

- Script modules offer an easy way to share functions and scripts with colleagues and peers
  - What sort of capabilities might you package into a script module for easier sharing?



### Key Points

Script modules offer an easy way to share functions and scripts with colleagues and peers. What sort of capabilities might you package into a script module for easier sharing?

## Lesson 3

# Writing Comment-Based Help

- Add comment-based help to a function
- Access a function's comment-based help from within the shell



As you produce more complex and reusable scripts and script modules, you want to include documentation with them so that both you and others can remember how to use them as time goes on. The built-in help system for Windows PowerShell, unfortunately, requires that help be defined in a fairly complex XML-based format. Fortunately, to add help to your scripts, functions, and script modules, you don't need to deal with that: you can simply add specially formatted comments right within the scripts or functions.

### Lesson Objectives

After completing this lesson, you will be able to:

- Add comment-based help to a function.
- Access a function's comment-based help from within the shell.

## Help

- You've already experienced the robust built-in help features that PowerShell and its cmdlets provide
  - Command syntax
  - Usage examples
  - Shell capabilities
- You can build "help" into your own scripts and functions that looks exactly like the shell's built-in help
  - Custom help does not require XML formatting
- Building help into your scripts makes their reusable code more easily-accessible to others
  - This is particularly the case for advanced functions because these functions behave much like real cmdlets

### Key Points

Administrators using Windows PowerShell rely on the shell's robust built-in help feature. The feature makes it easier to look up command syntax, obtain usage examples, and discover shell capabilities. You can build help into your scripts and functions that looks exactly like the built-in shell help—but you don't need to learn the complex XML formatting normally required to provide that help.

Providing this level of help for your scripts, functions, and script modules is easy, and doing so makes your reusable code more easily accessible to others. You should *especially* provide comment-based help when writing advanced functions. Because advanced functions behave so much like real cmdlets, others are likely to expect those advanced functions to have the same detailed help as a real cmdlet.

## Comment-Based Help

- Comment-based help can be added to scripts or functions
  - It uses a set of specially formatted comments to define various sections of the help.
  - It must appear as the first set of lines in the script or script module or immediately after the "function" keyword.
- Four commonly used help sections
  - SYNOPSIS – Provides a brief statement of what the script or function does.
  - DESCRIPTION – Provides a more detailed statement of what the script or function does.
  - PARAMETER – followed by a parameter name, describes the purpose of a single parameter. If you have more than one parameter, include a separate PARAMETER section for each one.
  - EXAMPLE – provides a usage example. You can have multiple EXAMPLE sections.

### Key Points

Comment-based help is normally added either to a script (or script module) or to a function. Typically, it makes the most sense to add it to a function because a function is a single, reusable unit.

Comment-based help, as the name implies, uses a set of specially formatted comments to define various sections of the help, such as parameters, usage examples, and so forth.

When used, comment-based help must be the first element within the script, script module, or function. It appears either as the first set of lines within the script or script module, or it is the first set of lines immediately after the "function" keyword.

You can construct the help as a series of one-line comments:

```
#.SYNOPSIS  
# This function creates a new user account in the domain. You must  
# run the function as an administrator.
```

Or you can use a block comment syntax:

```
<#  
.SYNOPSIS  
This function creates a new user account in the domain. You must  
run the function as an administrator.  
#>
```

Each section of the help starts with a period, then a section keyword, and then a carriage return. After the section keyword, you can have multiple lines of help text. Some of the most common section keywords include:

- SYNOPSIS—Provides a brief statement of what the script or function does.
- DESCRIPTION—Provides a more detailed statement of what the script or function does.

- **PARAMETER**—Followed by a parameter name, describes the purpose of a single parameter. If you have more than one parameter, include a separate PARAMETER section for each one.
- **EXAMPLE**—Provides a usage example. You can have multiple EXAMPLE sections.

**Note:** Other sections exist—run `help about_comment_based_help` for full details.

For example, a complete comment-based help block might look like this:

```
<#
.SYNOPSIS
Creates a new user in one of the company domains.
.DESCRIPTION
This function creates a new user in one of the company domains. The function fills in
all standardized information automatically. User-specific information is queried from
the personnel database—you simply need to provide the new user's e-mail address in order
to retrieve that information.
.PARAMETER emailAddress
The e-mail address of the new user, in the form first.last@division.contoso.com.
.PARAMETER kind
Provide Local, Remote, or Mixed to indicate the type of user.
.EXAMPLE
New-CompanyUser -emailAddress first.last@sales.contoso.com -kind Local
.EXAMPLE
The -kind parameter defaults to Local, so you do not need to provide it for local users:
New-CompanyUser -emailAddress first.last@sales.contoso.com
#>
```

When a user runs `help New-CompanyUser` (assuming that is the name of the function that the comment-based help is within), they will see a help screen that uses the same layout as the shell's built-in cmdlet help. Users could use the `-full`, `-detailed`, and `-examples` parameters to see variations of the help. For an advanced function, in which parameters can have attributes that make them mandatory, enable pipeline parameter binding, and so forth, the full help includes that information automatically.

**Note:** Comment-based help is especially useful within advanced functions. The shell's help system not only parses the comments to construct the help display, but it also pays attention to parameter attributes, whether or not the advanced function supports `ShouldProcess` and other elements. All this assistance enables the creation of a more complete and consistent help display.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Adding Comment-Based Help

- Learn how to add comment-based help to a script or function



### Key Points

In this demonstration, you will learn to add comment-based help to a script or function.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod08\Democode\Adding Comment-Based Help.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Discussion: Using Comment-Based Help

- Would comment-based help replace or supplement the kind of in-line comments that you usually expect to find in a script?
- Many organizations develop standards for in-line comments inside scripts or other programs
  - Would your organization develop similar standards for using comment-based help?



### Key Points

Would comment-based help replace or supplement the kind of inline comments that you usually expect to find in a script?

Many organizations develop standards for inline comments inside scripts or other programs; would your organization develop similar standards for using comment-based help?

## Lab: Advanced PowerShell Tips and Tricks

- Exercise 1: Writing a Profile Script
- Exercise 2: Creating a Script Module
- Exercise 3: Adding Help Information to a Function

Logon information

Virtual machine	LON-SVR1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 30 minutes**

### **Estimated time: 30 minutes**

You are a system administrator and have been developing lots of scripts and functions for several months. You want to pass this knowledge easily to your coworkers and also to provide them with good documentation on how to use what you have developed. You have also developed a few custom commands that you want to ensure are always available in your interactive shell. To accomplish this, you will add those commands to a profile script.

#### **Lab Setup**

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

1. Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Writing a Profile Script

### Scenario

You have a series of commands you want to run every time you start a new PowerShell console to save time.

The main tasks for this exercise are as follows:

1. Write a profile script.
2. Test a profile script.

#### ► Task 1: Write a profile script

- Write a profile script that does the following things:
  - a. Sets the current directory to C:.
  - b. Loads the Server Manager PowerShell module.
  - c. Prompts the user to enter alternate credentials and saves them to a variable.

#### ► Task 2: Test a profile script

- Load the previously written profile script into the current session.

**Results:** After this exercise, you will have written and tested a profile script that sets the current directory, loads the Server Manager module, and creates a variable containing a credential object.

## Exercise 2: Creating a Script Module

### Scenario

You have some long, one-line commands you often type to get status information on your servers and services. You want to add these often-typed commands into a module so that you can easily transfer them to other computers and easily share them with others. The main tasks for this exercise are as follows:

1. Package a function as a script module.
2. Load a script module.
3. Test the script module.

#### ► Task 1: Package a function as a script module

- Create the function **Get-AppEvents** and save it as a module script file name **MyAppEvents.psm1**.  
The function **Get-AppEvents** has the following code.

```
Function Get-AppEvents {  
    Get-EventLog -LogName Application -Newest 100 |  
        Group-Object -Property InstanceId |  
        Sort-Object -Descending -Property Count |  
        Select-Object -Property Name,Count -First 5 |  
        Format-Table -AutoSize  
}
```

**Hint:** Simply type the preceding code into a new script in the Windows PowerShell ISE. Then save the file as **MyAppEvents.psm1**. Be sure to save the file in the correct location for modules, in a module folder named **MyAppEvents**. For example:

`\Documents\WindowsPowerShell\Modules\MyAppEvents\MyAppEvents.psm1`

#### ► Task 2: Load a script module

- Load the script module just created named **MyAppEvents**.

#### ► Task 3: Test the script module

- Invoke the function **Get-AppEvents**.

**Results:** After this exercise, you will have written and tested a script module that runs a long command line that retrieves the five EventId values that have occurred the most often in the latest 100 events in the Windows Application event log.

## Exercise 3: Adding Help Information to a Function

### Scenario

You have some long, one-line commands that you have added to functions. You want to be able to use the Get-Help cmdlet to get help information from the functions you created to serve as a reminder on how to use them and what information they provide. The main tasks for this exercise are as follows:

1. Add comment-based help to a function.
2. Test comment-based help.
3. Reload a module.

#### ► Task 1: Add comment-based help to a function

- Edit the function Get-AppEvents in the MyAppEvents module created in Exercise 1 Task1 to add the following to the function, which will add a DESCRIPTION and EXAMPLE to the help information. For example, add the following:

```
<#
.SYNOPSIS
Get-AppEvents retrieves the newest 100 events from the Application
event log, groups and sorts them by InstanceID and then displays only
top 5 events that have occurred the most.
.DESCRIPTION
See the synopsis section.
.EXAMPLE
To run the function, simply invoke it:
PS> Get-AppEvents
#>
```

#### ► Task 2: Test comment-based help

- Test the comment-based help for the Get-AppEvents function with no parameters passed and also with the –Examples switch passed.

**Hint:** If no help is displayed, or if an error message is displayed, proceed to the next Task.

#### ► Task 3: Reload a module

- If the comment-based help is not working, remove and reimport the **MyAppEvents** module. Then test the comment-based help for the **Get-AppEvents** function again.

**Results:** After this exercise, you will have added and tested comment-based help in the function that retrieves the five EventId values that have occurred the most in the latest 100 events in the Windows Application event log.

## Lab Review

1. What's the difference between adding comment-based help to a function versus adding lots of regular comments in the script itself?
2. Why is it important to refrain from using customized aliases in a module that is shared with others?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- Can comment-based help include references to other help topics, or even to online content?
- Would you be more likely to use an all-hosts profile or a current-host profile?



### Review Questions

1. Can comment-based help include references to other help topics or even to online content?
2. Would you be more likely to use an all-hosts profile or a current-host profile?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 9

## Automating Windows Server® 2008 Administration

### Contents:

<b>Lesson 1:</b> Windows Server 2008 R2 Modules Overview	9-3
<b>Lesson 2:</b> Server Manager Cmdlets Overview	9-11
<b>Lab A:</b> Using the Server Manager Cmdlets	9-13
<b>Lesson 3:</b> Group Policy Cmdlets Overview	9-20
<b>Lab B:</b> Using the Group Policy Cmdlets	9-23
<b>Lesson 4:</b> Troubleshooting Pack Overview	9-29
<b>Lab C:</b> Using the Troubleshooting Pack Cmdlets	9-31
<b>Lesson 5:</b> Best Practices Analyzer Cmdlets Overview	9-36
<b>Lab D:</b> Using the Best Practices Analyzer Cmdlets	9-38
<b>Lesson 6:</b> IIS Cmdlets Overview	9-43
<b>Lab E:</b> Using the IIS Cmdlets	9-48

## Module Overview

After completing this module, you will be able to:

- List the modules included with Windows Server 2008 R2
- Add and remove Windows features from the command-line
- Manage Group Policy objects from the command-line
- Use Troubleshooting Packs from the command-line
- Run and review Best Practices Analyzer reports from the command-line
- Manage IIS features and configuration from the command-line



Windows Server® 2008 R2 ships with numerous Windows PowerShell® snap-ins and modules designed to help facilitate the administration of Windows Server 2008 R2 roles and features. Learning to use these modules and snap-ins provides you with the ability to automate additional Windows Server 2008 R2-related tasks, and also provides additional experience using Windows PowerShell itself, core cmdlets, and other techniques covered earlier in this course.

### Module Objectives

After completing this module, you will be able to:

- List the modules included with Windows Server 2008 R2.
- Add and remove Windows features from the command-line.
- Manage Group Policy objects from the command-line.
- Use Troubleshooting Packs from the command-line.
- Run and review Best Practices Analyzer reports from the command-line.
- Manage IIS features and configuration from the command-line.

## Lesson 1

# Windows Server 2008 R2 Modules Overview

After completing this lesson, you will be able to:

- List and describe the modules provided with Windows Server 2008 R2
- Import Windows Server 2008 R2 modules into the shell
- List the cmdlets and functions included in a given module



Sometimes, simply knowing that a tool is available to you can save you a great deal of time. That is particularly true with Windows PowerShell: Once you know that a tool or capability exists, discoverability features like Help, the Get-Member cmdlet, and so forth can help you learn how to use the tool or capability. The purpose of this lesson is to introduce you to the variety of modules and snap-ins available in Windows Server 2008 R2.

Note that some of these modules and snap-ins are also available in Windows® 7, particularly after installing the Remote Server Administration Toolkit (RSAT). Modules and snap-ins that are available in both Windows 7 and Windows Server 2008 R2 usually work identically—there are few or no functional differences just because a particular module is on the client or server operating system.

### Lesson Objectives

After completing this lesson, you will be able to:

- List and describe the modules provided with Windows Server 2008 R2.
- Import Windows Server 2008 R2 modules into the shell.
- List the cmdlets and functions included in a given module.

MCT USE ONLY. STUDENT USE PROHIBITED

## Modules Overview

- Windows Server 2008 R2's modules are installed in the PowerShell systemwide module folder. View the contents of this folder with:

```
Cd $pshome\modules
```

- PowerShell defines an environment variable with the search paths for locating modules. View the contents of this environment variable with:

```
Get-Content env:psmodulepath
```

- Generate a list of installed modules with:

```
Get-Module -list
```

- Generate a list of installed snap-ins with:

```
Get-PSSnapin -registered
```

## Key Points

Windows Server 2008 R2's modules are installed in the Windows PowerShell systemwide module folder. Within Windows PowerShell, you can run the following command to view the contents of the module folder:

```
Cd $pshome\modules
```

Windows PowerShell also defines an environment variable that contains the paths the shell will search to locate modules. You can view the contents of this environment variable by running this command:

```
Get-Content env:psmodulepath
```

You can also generate a list of installed modules by running this command:

```
Get-Module -list
```

Most of the Windows PowerShell extensions included with Windows Server 2008 R2 are provided as modules; some, however, are provided as snap-ins. To view a list of installed snap-ins, run this command:

```
Get-PSSnapin -registered
```

Note that not every module or snap-in is installed in Windows Server 2008 R2 by default. Typically, a module or snap-in is only installed when its associated role or feature is installed. For example, the ActiveDirectory module is normally installed when the Active Directory® Domain Services role is installed. So, in a typical production environment, you can expect some variation between the modules installed on different servers, because each server may have some variation in the roles and features installed on it.

Also, remember that you don't have to copy or install modules on every computer in your environment. Some modules, for example, may only run on Windows Server 2008 R2, and might not function when

copied to a Windows 7 or older client computer. That does not mean you can't use those modules from your Windows 7 or older client computer. One of the purposes of implicit remoting, which was covered in an earlier module of this course, is to enable you to use modules and snap-ins that only exist on a remote computer.

MCT USE ONLY. STUDENT USE PROHIBITED

## Included Modules

- The following modules and snap-ins are included with Windows Server 2008 R2:
  - Active Directory Domain Services (AD DS)
  - Active Directory Rights Management Services (AD RMS)
  - AppLocker (Application ID Policy Management)
  - Best Practices Analyzer
  - Background Intelligent Transfer Service (BITS)
  - Group Policy and Group Policy objects (GPOs)
  - Network Load Balancing
  - Windows PowerShell Diagnostics (PSDiagnostics)
  - Remote Desktop Services
  - Server Manager
  - Server Migration
  - Troubleshooting Packs
  - Web Administration (Internet Information Services)
  - Web Services for Management (WS-MAN)
  - Windows Cluster Service (Failover Clusters)
  - Windows Server Backup

## Key Points

The modules and snap-ins included with Windows Server 2008 R2 address the following features and roles:

- Active Directory Domain Services (AD DS)
- Active Directory Rights Management Services (AD RMS)
- AppLocker (Application ID Policy Management)
- Best Practices Analyzer
- Background Intelligent Transfer Service (BITS)
- Group Policy and Group Policy objects (GPOs)
- Network Load Balancing
- Windows PowerShell Diagnostics (PSDiagnostics)
- Remote Desktop Services
- Server Manager
- Server Migration
- Troubleshooting Packs
- Web Administration (Internet Information Services)
- Web Services for Management (WS-MAN)
- Windows Cluster Service (Failover Clusters)
- Windows Server Backup

In addition, some Windows Server 2008 R2 roles can be managed using third-party modules or snap-ins. For example, <http://pshyperv.codeplex.com/> is an open-source set of cmdlets designed to help manage Microsoft Hyper-V™ installations.

MCT USE ONLY. STUDENT USE PROHIBITED

## Working with Modules and Snap-ins

- Modules must be added into the shell to use their functionality. Modules remain for the duration of the shell session, unless they are specifically removed.

### **Import-Module ActiveDirectory**

- Once loaded, use Get-Command to list the cmdlets that are now available through the module.

### **Get-Command -module ActiveDirectory**

- Some Windows Server 2008 R2 extensions are not provided as modules. Instead they are snap-ins.
- Snap-ins require a slightly different syntax to load and list their contents.

**Add-PSSnapin windows.serverbackup  
Get-Command -pssnapin windows.serverbackup**

## Key Points

To add a module into the shell, use the Import-Module cmdlet and the name of the module. Note that modules remain loaded for the duration of your shell session, unless you remove them by running Remove-Module. You can see a list of currently-loaded modules by running Get-Module. Keep in mind that all modules are unloaded automatically when the shell is closed, and modules are not reloaded automatically. You can use Import-Module in a profile script (covered in the next modules of this course) to automatically load modules that you want to have available to you permanently.

For example:

```
Import-Module ActiveDirectory
```

Once a module is loaded, you can use Get-Command to list the cmdlets that were added by that module:

```
Get-Command -module ActiveDirectory
```

Some modules also add new PSDrive Providers. You can view all installed providers by running Get-PSProvider.

Some of the Windows Server 2008 R2 extensions—notably the Windows Server Backup extension—are not provided as modules. Instead, they are provided as snap-ins. Load these into the shell by using Add-PSSnapin and the full name of the snap-in. As with modules, these remain loaded for the duration of the shell session. To see the commands added by a snap-in, run:

```
Get-Command -pssnapin snap-in-name
```

Most extensions provided by Microsoft come with standard Windows PowerShell help files. Therefore, once a module or snap-in is loaded, you can use Get-Command to see what cmdlets the module or snap-

in has added to the shell. Once you know the cmdlet names, you can use Help to read help on those cmdlets, learn how to use them, and see examples of use.

MCT USE ONLY. STUDENT USE PROHIBITED

## About the Lessons in This Module

- This module includes five additional lessons:
  - Server Manager Cmdlets
  - Group Policy Cmdlets
  - Troubleshooting Pack Cmdlets
  - Best Practices Analyzer Cmdlets
  - IIS Cmdlets
- Each of the following lessons will briefly introduce the module being covered, and the details about the module's operation
  - You will complete a lab where you use the module in practice
  - The goal is for you to use the techniques here to identify the module, load it, list its cmdlets, and read help
  - Discover how to use the cmdlets in completing the labs

### Key Points

This module contains five additional lessons. Each lesson focuses on a single Windows Server 2008 R2 Windows PowerShell extension—primarily modules, rather than snap-ins.

Each lesson will briefly introduce the module being covered, and any details about the module's operation. Then you complete a lab in which you use the module to complete several tasks.

In general, the lessons will provide you with minimal information. The goal is for you to use the techniques outlined here, including cmdlets like Get-Module, Help, and Get-Command, to identify the module, load it into the shell, list its cmdlets, and read help on those cmdlets. By using the cmdlet help, and the examples included in the cmdlet help, you should be able to teach yourself how to use the cmdlets well enough to perform the tasks in the lab. Hints throughout the labs will remind you of key techniques to help you teach yourself what to do.

## Lesson 2

# Server Manager Cmdlets Overview

- Use Server Manager cmdlets to add roles, remove roles, and list roles
- Use Server manager cmdlets, along with Compare-Object, to provide a change configuration report for role changes on a server



The Server Manager cmdlets are designed primarily to list, install, and remove roles and features on the local Windows Server 2008 R2 computer. Keep in mind that you can access these cmdlets from a remote computer by using Windows PowerShell remoting, but the cmdlets are designed to execute locally on a Windows Server 2008 R2 computer.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use Server Manager cmdlets to add roles, remove roles, and list roles.
- Use Server manager cmdlets, along with Compare-Object, to provide a change configuration report for role changes on a server.

MCT USE ONLY. STUDENT USE PROHIBITED

## Server Manager Cmdlets

- The Server Manager module includes only three cmdlets:
  - Get-WindowsFeature
  - Add-WindowsFeature
  - Remove-WindowsFeature

**Import-Module servermanager**

### Key Points

The Server Manager module is fairly basic. It includes only three cmdlets, one of which generates a list of all available roles and features, indicating which ones are installed. The other two remove or add features.

The list of roles and features created by Get-WindowsFeature includes the official name of each role and feature. You will use this name along with

Add-WindowsFeature and Remove-WindowsFeature to add and remove roles and features.

Although the output of Get-WindowsFeature looks like a nicely formatted text list, it is in fact a set of objects, just like all other cmdlet output in the shell. A predefined view, included with the ServerManager module, formats those objects into the hierarchical list that is displayed on the screen. However, that formatting does not occur until the end of the pipeline. That means you can pipe those objects to other cmdlets to manipulate those objects or output them to a file of some kind.

**Note:** If you're curious about how modules are constructed, run  
cd \$pshome/modules/servermanager, then run gc servermanager.psd1, and then run ls. You will see the contents of the ServerManager module manifest, which references the module's binary file, its XML-based help file, and its .format.ps1xml view definition. The manifest is simply a list of module components, so that the shell knows to load them all. You will see the actual files in the directory listing generated by the ls command.

## Lab A: Using the Server Manager Cmdlets

- Exercise 1: Listing All Currently Installed Features
- Exercise 2: Comparing Objects
- Exercise 3: Installing a New Server Feature
- Exercise 4: Exporting Current Configuration to XML

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 20 minutes**

### Estimate time: 20 minutes

You work as a systems administrator, focusing on servers. You are configuring a new Windows Server 2008 R2 file server. The server must meet a standard configuration, and you must provide an audit trail of the final configuration in the form of an XML file.

#### Lab Setup

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

1. Start the **LON-DC1** virtual machine and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. On LON-DC1, open a Windows PowerShell session as **Administrator**.
3. On LON-DC1, open the **Start** menu. Under the **Administrative Tools** folder, open the **Group Policy Management Console**.
4. In the left pane of the Group Policy Management Console, expand the forest and domain until you see the **Starter GPO** folder. Select the **Starter GPO** folder by clicking on it.
5. In the right pane, click the **Create Starter GPOs** button to create the Starter GPOs.
6. Ensure that Windows PowerShell remoting is enabled on **LON-SVR1** and **LON-SVR2**. If you completed the previous labs in this course and have not shut down and discarded the changes in the virtual machines, remoting will be enabled. If not, open a Windows PowerShell session on those two virtual machines and run **Enable-PSRemoting** on each.
7. In Windows PowerShell, execute the following two commands:
  - **Set-ExecutionPolicy RemoteSigned**
  - **E:\Mod09\Labfiles\Lab\_09\_Setup.ps1**

MCT USE ONLY. STUDENT USE PROHIBITED

**Note:** Some of the Group Policy Objects and DNS entries that this script will modify may already exist in the virtual machine and you may receive an error saying so if you have used the virtual machine during earlier modules and have not discarded those changes.

8. Ensure that the Windows Firewall is disabled on **LON-SVR1**, **LON-DC1**, and **LON-SVR2**.
9. Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
10. Open a Windows PowerShell session as **Administrator**.

## Exercise 1: Listing All Currently Installed Features

### Scenario

You are configuring a new Windows Server 2008 R2 file server. Other administrators have been using it for testing, so you don't know what state it is in. You want to use the Server Manager cmdlets in Windows PowerShell to discover how the server is currently configured.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Import the Windows Server Manager PowerShell module.
2. View the module contents.
3. View all installed server features using Windows PowerShell.

► **Task 1: Import the Windows Server Manager PowerShell module**

1. On LON-SVR1, in Windows PowerShell, view the list of all available modules.
2. Import the **Server Manager** module into your PowerShell session.

► **Task 2: View the module contents**

- View the contents of the ServerManager module.

► **Task 3: View all installed server features**

1. Display the available and installed server features using Windows PowerShell and the Server Manager cmdlets.
2. Display only the server features that have been installed using Windows PowerShell and the Server Manager cmdlets.

**Note:** The Windows Feature object has a Boolean property called Installed that indicates whether it is installed or not.

**Results:** After this exercise, you should have successfully loaded the ServerManager PowerShell module, displayed the module contents, and viewed installed and available server features.

## Exercise 2: Comparing Objects

### Scenario

You need to compare the current configuration against your corporate standard. The standard configuration has been documented using the Export-Clixml cmdlet. You must compare the feature configuration information in this file against the current configuration.

The main tasks for this exercise are carried out on LON-SVR1 and as follows:

1. Import an XML file with standard configuration information.
2. Compare the standard configuration to the current configuration using the Compare-Object cmdlet.

► **Task 1: Import the XML file**

1. Import the **TargetFeatures.xml** file from the **E:\Mod09\Labfiles** folder, which was created with **Export-Clixml** cmdlet, as the variable **\$target**.
2. Create a variable that contains the current server feature configuration.

► **Task 2: Compare the imported features with the current features**

- Compare the installed property of the **\$target** and **\$current** objects using the **Compare-Object** cmdlet. The **\$target** variable is the reference object and **\$current** is the difference object.

**Note:** The installed property should be the only thing that is different, but you can include other property names to make the result more meaningful.

**Results:** After this exercise, you should be able to identify which server features need to be installed and which need to be removed to bring the configuration into compliance.

## Exercise 3: Installing a New Server Feature

### Scenario

To bring the server configuration into compliance, you need to install the missing server features.

The main task for this exercise is carried out on LON-SVR1 and is as follows:

- 1 Install the missing Windows Server features.

► **Task 1: View cmdlet help**

- View help and examples for the **Add-WindowsFeature** cmdlet.

► **Task 2: Install missing features**

- To install the missing features based on the comparison you did in Exercise 2, you could simply make a written note of features installed in **\$target**, but not installed in **\$current**, and manually use **Add-WindowsFeature** to install each one. You can also accomplish this task using Windows PowerShell. Write a pipelined expression to install features that should be installed but are not.

**Note:** Remember, you can use the `-WhatIf` common parameter with `Add-WindowsFeature` so you can preview what the cmdlet would do without actually making any changes.

**Results:** After this exercise, you should have installed the server features that were installed in the imported configuration but not in the current configuration.

## Exercise 4: Exporting Current Configuration to XML

### Scenario

To provide an audit trail, you must export the current configuration to an XML file.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Verify the current configuration of installed features.
2. Create an XML file with information about the current feature configuration.

► **Task 1: Verify the current configuration**

- Using Windows PowerShell, display only the currently installed features, and make sure only the following features are installed: **File-Services**, **FS-FileServer**, **Windows-Internal-DB**, **Backup-Features**, **Backup**, **Backup-Tools**, and **WSRM**.

► **Task 2: Export configuration to XML**

- Get all server features and export to a PowerShell XML file called **LON-SVR1-Features.xml**. Store the file in your documents folder.

**Results:** After this exercise, you should have created an XML file that contains the current and now standardized, feature configuration.

## Lab Review

1. What module do you need to import to manage server features?
2. What cmdlets are used to manage server features?
3. What type of object does Get-WindowsFeature write to the pipeline?
4. What are some of its properties?
5. Does the cmdlet support configuring a remote server?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 3

# Group Policy Cmdlets Overview

- Use Group Policy cmdlets to review Group Policy objects, create HTML reports, back up a Group Policy object, and more
- Use Group Policy cmdlets to create, link, and change settings within a GPO



The GroupPolicy module includes cmdlets designed to create, back up, restore, and manipulate Group Policy objects (GPOs) and GPO links within a domain.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use Group Policy cmdlets to review Group Policy objects, create HTML reports, back up a Group Policy object, and more.
- Use Group Policy cmdlets to create, link, and change settings within a GPO.

MCT USE ONLY. STUDENT USE PROHIBITED

## Group Policy Cmdlets

- The Group Policy module includes 25 cmdlets
  - Backup-GPO
  - Copy-GPO
  - Get-GPIInheritance
  - Get-GPO
  - Get-GPReport
  - Get-GPPermissions
  - Get-GPPrefRegistryValue
  - Get-GPRegistryValue
  - Get-GPResultantSetOfPolicy
  - Get-GPStarterGPO
  - Import-GPO
  - New-GPLink
  - New-GPO
  - New-GPStarterGPO
  - Remove-GPLink
  - Remove-GPO
  - Remove-GPPrefRegistryValue
  - Remove-GPRegistryValue
  - Rename-GPO
  - Restore-GPO
  - Set-GPIInheritance
  - Set-GPLink
  - Set-GPPermissions
  - Set-GPPrefRegistryValue
  - Set-GPRegistryValue

### Key Points

Remember that a Group Policy consists of two distinct components:

- A GPO file that resides on disk, containing the settings of the GPO.
- A link between the GPO and a container, site, organizational unit, or domain. The link determines which users and computers are affected by the GPO.

A complete discussion of how GPOs work is beyond the scope of this course, although as an experienced Windows Server administrator you should already have some familiarity with their operation.

Three primary cmdlets exist to work with GPOs:

- **Get-GPO:** Retrieves an existing GPO, usually by name.
- **New-GPO:** Creates a new GPO.
- **Remove-GPO:** Removes a GPO.

Other cmdlets can rename, copy, back up, and restore a GPO.

Cmdlets designed to work with GPO links include:

- New-GPLink
- Remove-GPLink
- Set-GPLink

Other cmdlets exist to work with Group Policy inheritance settings, to work with resultant set of policy (RSOP), and to work with individual settings within a GPO. For example, Set-GPPrefRegistryValue and Set-GPRegistryValue enable you to change individual settings within a GPO.

**Note:** Remember, read the help on these cmdlets, and use help with the –full parameter, to see examples of use.

The pattern for working with GPOs from within the shell is very similar to the workflow you would use in the Group Policy Management Console.

1. Start by getting or creating an existing or new GPO.
2. Change one or more settings or registry settings (*preferences*) within the GPO.
3. Link the GPO to one or more locations within the domain.

Recalling the workflow that you would use in the graphical console can help you understand the sequence of commands you would run to accomplish the same tasks from the command-line.

Take some time to review the cmdlets added by the GroupPolicy modules. Other cmdlets can produce reports in HTML format, manage GPO permissions, and much more. Read the help for these cmdlets and examples of use to understand what they do, and what options they offer for customizing their behavior.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab B: Using the Group Policy Cmdlets

- Exercise 1: Listing All the Group Policy Objects in the Domain
- Exercise 2: Creating a Text-Based Report
- Exercise 3: Creating HTML Reports
- Exercise 4: Backing Up All Group Policy Objects

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 20 minutes**

### Estimated time: 20 minutes

You are a Windows administrator responsible for managing Group Policy objects in your domain. Your manager has asked for a report of all your Group Policy Objects. Your manager also wants to make sure that all Group Policy Objects are being backed up.

#### Lab Setup

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

1. Start the **LON-SVR1** virtual machine. You do not need to log on, but wait until the boot process is complete.
2. Start the **LON-DC1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open a Windows PowerShell session.

## Exercise 1: Listing All Group Policy Objects in the Domain

### Scenario

Your manager has asked for a report of all your Group Policy Objects. You will start this process by gathering a list of all the Group Policy Objects in the domain.

The main tasks for this exercise are carried out on LON-DC1 and are as follows:

1. Import the Group Policy PowerShell module.
2. View the module contents.
3. View all Group Policy objects using Windows PowerShell.

► **Task 1: Import the Group Policy PowerShell module**

1. Open Windows PowerShell and view the list of available modules.
2. Import the **Group Policy** module into your PowerShell session.

► **Task 2: View the module contents**

- View the contents of the Group Policy module.

► **Task 3: View Group Policy objects**

- Display all Group Policy objects in the domain using Windows PowerShell.

**Results:** After this exercise, you should have successfully loaded the GroupPolicy PowerShell module, displayed the module contents, and viewed all Group Policy objects in the domain.

## Exercise 2: Creating a Text-Based Report

### Scenario

Your manager has asked for some brief, summary reports of the Group Policy objects in your domain in a text file format.

The main tasks for this exercise are as follows:

1. Get selected information from all Group Policy objects in the domain.
2. Create a text file with this information.

#### ► Task 1: Get selected Group Policy object properties

- Using Windows PowerShell, get all Group Policy objects for the domain displaying the GPO name, when it was created, and when it was last modified. The output should be sorted by the time the GPO was last modified.

**Note:** Group Policy objects in PowerShell are like any other object in that they have properties. Use Get-Member to discover property names.

#### ► Task 2: Create a text file report

- Collect the same information specified in the previous task and save the results to a text file called **GPOSummary.txt** in **\LON-SVR1\E\$\Mod09\Labfiles**.

**Results:** After this exercise, you should have a text file that displays selected properties about all Group Policy objects in your domain.

## Exercise 3: Creating HTML Reports

### Scenario

You need to document Group Policy objects for historical and audit purposes. Your manager has also asked for a HTML-formatted report on the default domain policy.

The main tasks for this exercise are as follows:

1. Create an HTML report for the Default Domain policy.
2. Create an individual HTML report for each group policy object.

#### ► Task 1: Create a default Domain Policy report

- Using Windows PowerShell, create an HTML report for the default domain policy called **DefaultDomainPolicy.htm** and save it in your documents folder.

#### ► Task 2: Create individual GPO reports

- Using a PowerShell expression, create individual HTML reports for each Group Policy object and save them to **\LON-SVR1\E\$\Mod09\Labfiles**. The filename should be in the format *GPO display name.htm*.

**Hint:** You need to retrieve the GPOs, and then enumerate them by using `ForEach-Object`. Within the `ForEach-Object` script block, use a GPO cmdlet that generates a GPO report. That cmdlet identifies the GPOs by using an `-id` parameter; using `$_.id` for that parameter value will identify each GPO that you enumerate.

**Results:** After this exercise, you should have html reports for every Group Policy object in the domain.

## Exercise 4: Backing Up All Group Policy Objects

### Scenario

For business continuity purposes it is critical that Group Policy objects are properly backed up on a weekly basis. Your manager has asked you to develop a PowerShell solution you can run on a weekly basis to back up all Group Policy objects.

The main task for this exercise is:

- 1 Back up a Group Policy Object individually and collectively.

► **Task 1: Back up the Default Domain policy**

- Using Windows PowerShell, create a local backup of the Default Domain policy to your documents folder.

► **Task 2: Back up all Group Policy objects**

- Using Windows PowerShell, back up all group policy objects in the domain to **\LON-SVR1\E\$\Mod09\Labfiles**. Add the comment *Weekly Backup* to each backup.

**Results:** After this exercise, you should have backups of every Group Policy object in the domain.

## Lab Review

1. What module do you need to import to manage Group Policy?
2. What cmdlet is used to get Group Policy information?
3. What type of Group Policy reports can you create?
4. Can you back up individual Group Policy objects?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 4

# Troubleshooting Pack Overview

- Execute Troubleshooting Packs on a local or remote computer by using one-to-one interactive remote shell or implicit remoting



The TroubleshootingPack module includes cmdlets that run Windows Troubleshooting Packs from the command-line. These cmdlets not only provide a useful way of automating Troubleshooting Pack execution, but also the ability to, through Windows PowerShell Remoting, connect to remote server and client computers and run Troubleshooting Packs on them.

Troubleshooting Packs are a new feature of Windows 7 and Windows Server 2008 R2 that provide interactive problem troubleshooting, diagnosis, and repair for key problem areas.

### Lesson Objectives

After completing this lesson, you will be able to:

- Execute Troubleshooting Packs on a local or remote computer by using one-to-one interactive remote shell or implicit remoting.

MCT USE ONLY. STUDENT USE PROHIBITED

## Troubleshooting Pack Cmdlets

- The Troubleshooting Pack module includes two cmdlets
  - Get-TroubleshootingPack
  - Invoke-TroubleshootingPack
- These cmdlets require Troubleshooting Packs, most of which are installed by default to `\Windows\Diagnostics\System`

**Get-TroubleshootingPack**  
`C:\windows\diagnostics\system\Aero`

- Many Troubleshooting Packs are interactive at the shell, presenting a command-line menu for selecting specific problems or diagnostics

### Key Points

The TroubleshootingPack module provides only two cmdlets:

- `Get-TroubleshootingPack` loads a troubleshooting pack from disk.
- `Invoke-TroubleshootingPack` executes a loaded troubleshooting pack.

To load a troubleshooting pack, you need to know its path. Most troubleshooting packs are installed in `\Windows\Diagnostics\System`, so you could run a command like this:

```
Get-TroubleshootingPack c:\windows\diagnostics\system\Aero
```

You can view a directory of `c:\windows\diagnostics` to see troubleshooting packs that are installed on a computer. Note that there will be some variation in installed troubleshooting packs based on the computer's operating system version, installed roles and features, and so on.

Once you load a troubleshooting pack, you can pipe it to `Invoke-TroubleshootingPack` to begin the troubleshooting process. Many troubleshooting packs are interactive, and present a command-line menu that you can use to select specific problems or diagnostics.

Troubleshooting packs can be automated by means of an XML-based answer file. Review the examples in the help file for `Get-TroubleshootingPack` to see how to generate a blank answer file for a given troubleshooting pack. After customizing the answer file as desired (using Windows Notepad or the Windows PowerShell ISE), you can use the `-unattend` and `-answer` parameters of `Invoke-TroubleshootingPack` to specify the answer file and run diagnostics in unattended mode.

A good scenario for using troubleshooting packs is to use Windows PowerShell 1-to-1 remoting to obtain a shell prompt on a remote computer. This enables you to execute the troubleshooting packs available on that remote computer, even while a user is interactively logged on to that computer. This technique allows you to troubleshoot and resolve problems for remote users.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab C: Using the Troubleshooting Pack Cmdlets

- Exercise 1: Importing the Troubleshooting Pack Module
- Exercise 2: Solving an End-User Problem Interactively
- Exercise 3: Solving a Problem Using Answer Files

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-CLI1
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 20 minutes**

### Estimated time: 20 minutes

You work as a systems technician, troubleshooting problems with individual desktops. The help desk manager has asked for your assistance in troubleshooting a desktop problem. She has heard you can solve problems with Windows PowerShell and has asked for your assistance in learning more.

#### Lab Setup

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

- 1 Start the **LON-DC1** and **LON-SVR1** virtual machines. You do not need to log on, but wait until the boot process is complete.
- 2 Start the **LON-CLI1** virtual machine, and then logon by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 3 Open a Windows PowerShell session on LON-CLI1.

## Exercise 1: Importing the Troubleshooting Pack Module

### Scenario

Your company has standardized on Windows 7 as the client operating system which includes PowerShell. You need to become familiar with PowerShell-based tools that you can use to troubleshoot and resolve problems.

The main tasks for this exercise are carried out on LON-CLI1 and are as follows:

1. Import the Troubleshooting Pack PowerShell module.
2. View the module contents.

► **Task 1: Import the Troubleshooting Pack PowerShell module**

- On LON-CLI1, import the **Troubleshooting Pack** module into your PowerShell session.

► **Task 2: View the module contents**

- View the contents of the Troubleshooting Pack module.

**Results:** After this exercise, you will have successfully loaded the Troubleshooting Pack module and know what cmdlets it offers.

## Exercise 2: Solving an End-User Problem Interactively

### Scenario

Since you have to support Windows 7 clients, you plan to use Windows PowerShell to resolve problems faster and more efficiently. There is an ongoing problem with the Windows Search service. Using the Troubleshooting Pack cmdlets, your job is to find a way to identify this problem and resolve it.

The main tasks for this exercise are:

1. Stop Windows search.
2. Run an interactive troubleshooting session.
3. View the report.

► **Task 1: Stop Windows search**

- To simulate the problem, use Windows PowerShell to stop the Windows Search service if it is running.

► **Task 2: Run an interactive troubleshooting session**

1. Using the Search troubleshooting pack, run a troubleshooting session and save the results to the current directory. When prompted, indicate that *Files don't appear in search results*.
2. When prompted, allow the troubleshooter to fix the problem.

► **Task 3: View the report**

- Using Internet Explorer, view the contents of ResultReport.xml.

**Results:** After this exercise, you should have successfully run an interactive troubleshooting session in Windows PowerShell and generated a report file.

## Exercise 3: Solving a Problem Using Answer Files

### Scenario

You have a recurring problem that is relatively easy to resolve once it has been identified. You want to develop an answer file for this problem so that if the suspected problem exists, you can run the file so, the problem can be resolved quickly with minimal intervention on your part.

The main tasks for this exercise are as follows:

1. Create a troubleshooting pack answer file.
2. Use the answer file to resolve a problem.
3. Run a troubleshooting session unattended.

#### ► Task 1: Create an answer file

- Create an answer file called **SearchAnswer.xml** in your documents folder for the Search troubleshooting pack. When prompted, answer that *Files Don't Appear in Search Results*. Leave any other questions blank.

#### ► Task 2: Invoke troubleshooting using the answer file

1. Using Windows PowerShell, stop the Windows Search service if it is running. This will simulate the problem.
2. Use the answer file you created in Task 1 and invoke the Search troubleshooting pack. Do not fix any problems when prompted.

#### ► Task 3: Invoke troubleshooting unattended

- Using the answer file you created in Task 1, invoke the Search troubleshooting pack in unattended mode.

#### ► Task 4: Verify the solution

- Using Windows PowerShell, check the status of the Windows Search service and verify that it is now running.

**Results:** After this exercise, you should have a troubleshooting answer file which you have used interactively and unattended to troubleshoot and solve a problem.

## Lab Review

1. What cmdlet do you use to load the Troubleshooting Pack?
2. What are the cmdlets in the Troubleshooting Pack?
3. Can you resolve a problem without using an answer file?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 5

# Best Practices Analyzer Cmdlets Overview

- Generate Best Practices Analyzer reports by using the BPA cmdlets



The BestPractices module includes cmdlets designed to execute and review Best Practices Analyzer (BPA) reports.

The BPA is a feature of Windows Server 2008 R2 that is designed to review the configuration of the roles and features installed on a server, and compare those configurations to known Microsoft-defined best practices. By using the BPA cmdlets to automate the creation and review of BPA reports, you can help keep your servers configured for better security and reliability.

### Lesson Objectives

After completing this lesson, you will be able to:

- Generate Best Practices Analyzer reports by using the BPA cmdlets.

## BPA Cmdlets

- The Best Practice Analyzer module includes four cmdlets:
  - Get-BpaModel
  - Get-BpaResult
  - Invoke-BpaModel
  - Set-BpaResult
- BPA cmdlets work with BPA Models. Each model is a set of best practices that are related to a role or feature
  - Models must be retrieved and then invoked before their results can be retrieved

**Get-BpaModel | Invoke-BpaModel**

### Key Points

The BPA cmdlets work with *BPA Models*. Each model is a set of best practices related to a particular server role or feature. Running **Get-BpaModel** with no parameters produces a list of available models. This list varies depending upon the server roles and features installed on the server. The list display includes the official ID for each model. You need this ID in order to retrieve a specific model for execution.

When you retrieve one or more specific models, you can pipe them to **Invoke-BpaModel** to execute those models and generate a BPA report.

BPA reports are not immediately displayed. Instead, they are stored in the operating system and must be retrieved. To do so, use the **Get-BpaResult** cmdlet, specifying the ID of the BPA model for which you want to retrieve results. There will be no results available, of course, until the given BPA model has been invoked.

Results are not returned as a text listing. Like almost everything else in the shell, BPA results are collections of objects that you can sort, filter, and so on—including exporting them to a file or converting them to other formats such as HTML.

The help files for BPA cmdlets contain detailed instructions for using them, along with examples of use.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab D: Using the Best Practices Analyzer Cmdlets

- Exercise 1: Importing the Best Practice Module
- Exercise 2: Viewing Existing Models
- Exercise 3: Running a Best Practices Scan

Logon information

Virtual machine	LON-DC1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 15 minutes**

### Estimated time: 15 minutes

You work as a systems administrator, and you are responsible for a group of servers running Windows Server 2008 R2. As part of your automated audit process, you want to prepare a series of best practice reports depending on what server roles are installed.

#### Lab Setup

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

- 1 Start the **LON-DC1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 2 Open a Windows PowerShell session.

## Exercise 1: Importing the Best Practices Module

### Scenario

As a Windows administrator, it is very important that your systems operate at peak efficiency while remaining secure. As a company policy you closely follow Microsoft recommended best practices. Windows PowerShell has a solution for analyzing your computers and network to see if they meet their recommended best practices. You want to learn more about this.

The main tasks for this exercise are as follows:

1. Import the Best Practices PowerShell module.
2. View the module contents.

► **Task 1: Import the Best Practices PowerShell module**

1. Open Windows PowerShell and view the list of available modules.
2. Import the **Best Practices** module into your PowerShell session.

► **Task 2: View the module contents**

- View the contents of the Best Practices module.

**Results:** After this exercise, you should have successfully loaded the Best Practices PowerShell module and displayed the module contents.

## Exercise 2: Viewing Existing Models

### Scenario

In continuing your investigation of how PowerShell can help with analyzing best practices, you need to discover what areas it can scan.

The main task for this exercise is:

- 1 Display all available best practice models.

► **Task 1: Display all available best practice models**

- In Windows PowerShell, display all the best practice models you can use.

**Results:** After this exercise, you should be able to identify which areas can be scanned for best practices.

## Exercise 3: Running a Best Practices Scan

### Scenario

Now that you know what areas you can scan, you decide to discover how well your systems match up to the best practice model.

The main tasks for this exercise are as follows:

1. Run a best practices scan.
2. View the results.
3. Create a best practices scan report.

► **Task 1: Run a best practices scan**

- Using Windows PowerShell, run a best practices scan for Active Directory.

► **Task 2: View the scan results**

- Save the results of the Active Directory scan to a variable called **\$results** and view the results.

► **Task 3: View selected results**

- Each item in a scan result has a **Severity** property. Using Windows PowerShell, display only scan results from your last Active Directory scan that are of the Error type.

► **Task 4: Create a best practices report**

1. Using Windows PowerShell, run a best practices scan for Active Directory and generate an HTML report capturing these properties: **ResultID**, **Severity**, **Category**, **Title**, **Problem**, **Impact**, **Resolution**, and **Compliance**. The report should be sorted by **Severity**. Save the file to your documents folder. Use the CSS file found in your lab folder to provide formatting.
2. Open Windows Internet Explorer® to view the file.

**Results:** After this exercise, you should have an HTML report that compares your current Active Directory configuration to the Best Practices model.

## Lab Review

1. What module do you need to import to run Best Practices scans?
2. What are some of the items that you can view in a scan result?
3. What are some of the Best Practice models available?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 6

# IIS Cmdlets Overview

- Use IIS cmdlets to manage IIS features and Web sites
- Use the IIS: drive to manage IIS settings



The WebAdministration module includes cmdlets designed to manage Internet Information Services (IIS) objects, including Web sites and application pools. The module also includes a PSDrive provider, which automatically maps an IIS: drive when the module is loaded. The IIS: drive is used to manage many of the granular configuration settings within IIS.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use IIS cmdlets to manage IIS features and Web sites.
- Use the IIS: drive to manage IIS settings.

MCT USE ONLY. STUDENT USE PROHIBITED

## IIS Cmdlets

- The IIS module includes 74 cmdlets, which are intended to represent the core set of administrative actions that one would use with an IIS server and websites
  - Creating Web sites
  - Creating FTP sites
  - Creating Application pools
  - Start, stop, restart sites
  - Start, stop, restart pools
- The cmdlets in the IIS module contain excellent help files and usage examples

**Import-Module WebAdministration**

### Key Points

The WebAdministration module includes 74 cmdlets, most of which perform high-level IIS tasks such as creating Web sites, creating FTP sites, creating application pools, and so on. They also enable you to start, stop, and restart sites and pools. The help files for the cmdlets contain detailed information on their usage, along with ample usage examples.

However, much of the IIS configuration within Windows PowerShell is not accomplished through these cmdlets. Instead, the model created by the IIS product team relies equally on the use of the IIS: drive.

## The IIS: Drive

- Due to the vast differences between IIS instances, IIS cmdlets are not comprehensive.
- Much of the IIS PowerShell administration occurs using the IIS: drive. This drive looks and behaves in similar ways to other PS drives (such as disk, Active Directory).
  - Each drive is actually a configuration repository for an area of the server/website.
  - Exposed in a hierarchical fashion.
  - Navigated and manipulated using the -Item and -ChildItem cmdlets.

```
PS IIS:\> cd apppools  
PS IIS:\apppools> ls
```

Name	State	Applications
-----	-----	-----
Classic .NET AppPool DefaultAppPool	Started Started	Default Web Site

### Key Points

One of the key facts about IIS 7.5 (the version included with Windows Server 2008 R2) is that it has a modular architecture. That means additional functionality can be added to IIS by installing plug-ins and add-ons, either through the Server Manager console or by downloading and installing add-ons (many are available from the <http://iis.net> Web site).

Because every IIS installation may be slightly different, with different add-ons and capabilities, it would be difficult to create a single set of cmdlets capable of managing every possible configuration permutation. Instead, the IIS development team created cmdlets to handle the core, fixed IIS elements that every IIS server supports: Application pools, Web sites, FTP sites, and so on.

The detailed configuration of the server and its sites, pools, and other elements is done through a hierarchical IIS: drive, which looks and behaves in many ways like a local hard disk or the registry drives that are mapped in Windows PowerShell.

For example, the top level of the IIS drive contains these folders:

```
AppPools  
Sites  
SslBindings
```

Changing to the AppPools folder reveals the configured application pools, such as this:

```
PS IIS:\> cd apppools  
PS IIS:\apppools> ls
```

Name	State	Applications
-----	-----	-----
Classic .NET AppPool DefaultAppPool	Started Started	Default Web Site

You can then change directories into one of these application pools, using this command:

```
PS IIS:\appools> cd defaultapppool  
PS IIS:\AppPools\defaultapppool> ls
```

Name
-----
WorkerProcesses

You can continue to navigate the configuration hierarchy in this way. Note that you are not using an actual disk; instead, you are accessing a configuration repository that is organized in a disk-like, hierarchical fashion. The PSDrive provider included with the WebAdministration module handles the work of presenting the configuration repository in a disk-like metaphor.

You use the shell's `-Item` and `-ChildItem` cmdlets to navigate the configuration repository, retrieve existing configuration values, and make changes to configuration values.

## Demonstration: IIS Examples

- Learn how to use the IIS: drive to review and manage IIS configuration settings from within the shell



### Key Points

In this demonstration, you will learn how to use the IIS: drive to review and manage IIS configuration settings from within the shell.

### Demonstration steps:

- 1 Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
- 2 Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod09\Democode\IIS Examples.ps1**.
- 3 Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab E: Using the IIS Cmdlets

- Exercise 1: Importing the IIS Module
- Exercise 2: Creating a New Web Site
- Exercise 3: Backing Up IIS
- Exercise 4: Modifying Web Site Bindings
- Exercise 5: Using the IIS PSDrive
- Exercise 6: Restoring an IIS Configuration

Logon information

Virtual machine	LON-DC1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are a Windows Administrator tasked with managing a small Web server used for internal purposes. Your job includes creating and managing Web sites. Because your company charges for internal IT services, you need to automate Web site management so that it can be done in the most efficient manner possible.

#### Lab Setup

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

- 1 Start the **LON-DC1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
- 2 Open a Windows PowerShell session.

## Exercise 1: Importing the IIS Module

### Scenario

Your job includes creating and managing Web sites. Because your company charges back for IT services, you need to automate Web site management so that it can be done in the most efficient manner possible. You have heard that you can manage IIS from Windows PowerShell and want to learn more.

The main tasks for this exercise are as follows:

1. Import the IIS PowerShell module.
2. View the module contents.

► **Task 1: Import the IIS PowerShell module**

1. Open Windows PowerShell and view the list of available modules.
2. Import the IIS management module into your PowerShell session.

► **Task 2: View the module contents**

- View the contents of the IIS module.

**Results:** After this exercise, you should have successfully loaded the IIS PowerShell module and displayed the module contents.

## Exercise 2: Creating a New Web Site

### Scenario

You want to test the IIS cmdlets by creating a simple Web site.

The main tasks for this exercise are as follows:

1. Create a simple Web site using the IIS cmdlets.
2. View all web sites.
3. Verify the Web site.

#### ► Task 1: Create a new Web site

1. Using the IIS cmdlets, create a new Web site using these settings:
  - a. Name: **PowerShellDemo**
  - b. Port: **80**
  - c. Physical Path: **%SystemDrive%\inetpub\PowerShellDemo**
2. Copy underconstruction.htm from the lab folder to the root of the new site as **default.htm**.

#### ► Task 2: View all Web sites

- Using Windows PowerShell and the IIS cmdlets, view all Web sites running on LON-DC1.

#### ► Task 3: Verify the Web site

1. Using Windows PowerShell, stop the Default Web site if it is running.
2. Using Windows PowerShell, start the new PowerShellDemo site.
3. Open the LON-DC1 Web server in Internet Explorer and verify you have reached the new site.

**Results:** After this exercise, you should be able to open the new Web site in Internet explorer and view the default document.

## Exercise 3: Backing Up IIS

### Scenario

Before making changes to IIS or Web site configurations, you need to have a backup. You want to use the IIS cmdlets to back up your IIS configuration.

The main tasks for this exercise are as follows:

1. Back up your IIS configuration.
2. Verify the backup.

► **Task 1: Back up IIS**

- Using the IIS cmdlets, back up your IIS configuration with a name of **Backup1**.

► **Task 2: Verify the backup**

- Using the IIS cmdlets, verify the Backup1 backup you just created.

**Results:** After this exercise, you should have a verified backup of IIS.

## Exercise 4: Modifying Web Site Bindings

### Scenario

To provide additional security for your internal Web server, you want to change the binding from the default port to a new port.

The main tasks for this exercise are as follows:

1. Display the current binding information for all Web sites.
2. Change the binding information for all sites to a new port.
3. Verify the site bindings have been changed.

► **Task 1: Display current binding information for all Web sites**

- Using the IIS cmdlets, display current binding settings for all Web sites on LON-DC1.

► **Task 2: Change port binding**

- Using Windows PowerShell, change the port binding for all Web sites on LON-DC1 from port **80** to port **8001**. This includes the site using a host header.

► **Task 3: Verify the new bindings**

1. Using Windows PowerShell and the IIS cmdlets, view the new binding information.
2. Using Internet Explorer, connect to the PowerShell Demo site on LON-DC1 using the new binding information.

**Results:** After this exercise, you should have changed the default binding on all sites to 8001 and verified the change.

## Exercise 5: Using the IIS PSDrive

### Scenario

Because your Web server includes several sites, you do not want the default site to start automatically. Using the IIS PSDrive, you need to reconfigure the site so that it does not start automatically.

The main tasks for this exercise are as follows:

1. Use the IIS PSDrive to navigate through the site folder.
2. Display current properties for the default site.
3. Modify the default site so that it is not automatically started by the server.

► **Task 1: Connect to the IIS PSDrive**

1. Display all available PSDrives in your PowerShell session.
2. Change your PowerShell location to the root of your IIS PSDrive.

► **Task 2: Navigate to the default site**

- Within the IIS PS Drive, navigate to the folder that represents the Default Site.

► **Task 3: Display all properties for the default site**

- Using Windows PowerShell and the IIS PSDrive, display a list of all properties for the default site.

**Tip:** Not all properties may be displayed by default.

► **Task 4: Modify the default site**

1. Using Windows PowerShell and the IIS PSDrive, modify the default site so that the site does not start automatically.
2. Change your PowerShell location back to C:\.

**Results:** After this exercise, you should have navigated through the IIS PSDrive and modified the Default Web site so that it does not start automatically.

## Exercise 6: Restoring an IIS Configuration

### Scenario

You realize you don't want to keep the changes you've made so you wish to restore IIS to its previous configuration.

The main tasks for this exercise are:

1. View all available IIS configuration backups.
2. Restore the most current IIS backup.

► **Task 1: View existing backups**

- Using Windows PowerShell and the IIS cmdlets, display all available IIS configuration backups.

► **Task 2: Restore the most recent configuration**

- Using Windows PowerShell and the IIS cmdlets, restore the most recent IIS configuration.

**Results:** After this exercise, you should have restored IIS back to an earlier configuration with all sites back on port 80. You can ignore any errors about denied access to c:\windows\system32\inetserv\config\scheman\wsmnconfig\_schema.xml.

## Lab Review

1. What module do you need to import to manage IIS?
2. What cmdlet do you use to create a new Web site?
3. What cmdlet do you use to stop a Web site?
4. What cmdlets do you use to modify Web site bindings?
5. What do you see in the Sites folder of the IIS PSDrive?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 10

## Reviewing and Reusing Windows PowerShell® Scripts

### Contents:

Lesson 1: Example Script Overview	10-3
Lesson 2: Understanding Scripts	10-8

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Overview

- Review a complete, real-world script
- Learn selected scripting constructs from the examples in the completed script
- Learn how to develop an expectation of what an existing script will do by reviewing the script



One of the main ways in which you will use Windows PowerShell® is to run scripts written by others. These scripts may be written by someone within your organization, or by individuals from outside your organization. You might, for example, obtain scripts from an online script repository, from a book, training video, and so on.

Therefore, a key skill in Windows PowerShell is being able to read scripts, interpret the commands in them, and develop an expectation of what the script does and how it works. Doing so can also be a good way to learn more advanced Windows PowerShell features, such as scripting constructs.

In this module, you will review a completed example script, and use that script as a way to learn selected scripting constructs.

### Module Objectives

After completing this module, you will be able to:

- Review a completed script and identify key functionality.
- Identify and explain the use of selected scripting constructs.

## Lesson 1

# Example Script Overview

- Quickly review the example script and understand the script pieces that comprise the example



One of the first things you should do when you acquire a new script is to review it and identify what the script does and how it works. This is accomplished by identifying major functional areas within the script. You can also identify specific areas that require customization in order for the script to be compliant with your environment.

### Lesson Objectives

After completing this lesson, you will be able to:

- Review an example script and identify how it works and what its major functional elements are.
- Identify major areas of the script that may require customization to work within your environment.

MCT USE ONLY. STUDENT USE PROHIBITED

## Reviewing the Example Script

- The example script actually consists of seven individual script files
- One of those files is the “main” script that incorporates the others and begins the main task that the script completes
- The seven files create one large script, which is a new user provisioning script that includes several different provisioning tasks
  - Creates new users in Active Directory
  - Adds users to groups
  - Creates new virtual directories in IIS
  - Creates new home folders
  - Applies permissions to home folders
  - Copies a set of default documents into home folders

### Key Points

The example script is a new user provisioning script. It is designed to complete several tasks, including creating new users in Active Directory®, adding those users to user groups, creating new IIS “home page” Web sites, new home folders, and so on.

Most of the functionality is contained in six individual, task-focused scripts. A seventh script initiates provisioning and coordinates the activities of the other six.

## Demonstration

- Let's briefly look at the example script and try to understand its purpose and methodology

### Key Points

Remember that our goal now is not to understand the entire set of scripts in detail; rather, our goal is to understand the script's general purpose and functionality, and to identify areas of the script that may require customization to work in a specific environment.

Begin by opening E:\Mod10\democode\Provision.ps1 in the Windows PowerShell ISE.

Line 5 contains useful information: These parameters are the pieces of information that the script needs in order to run. It needs a filename, which is the \$file parameter, and the text indicates that it expects the name of a .CSV file. It also needs a \$log parameter, but provides a default value for that parameter.

Lines 9 through 14 are dot sourcing other scripts. That means those scripts will be a part of this one, and so we will need to examine each of them as well.

At this point, we want to begin developing an idea of what the script does, and what might need to be modified in order for the script to run in a particular environment. The key is to focus on the commands that are included in the script.

One such command is on line 33, which imports a .CSV file. This is using the \$file parameter for the filename, which means we don't need to customize this to make the script work. Instead, we provide the desired filename to the script, by means of the \$file parameter, when we run the script.

Line 43 has the New-User command. Note that this is *not* the New-ADUser cmdlet—New-User appears to be a custom command or function. We will have to look at those other six scripts to see if New-User resides in one of them. Note that there is information here that will need to be customized: Organizational unit and company, for example. This also gives us our first clue about what this script does—it creates new users.

Line 61 is another command, Add-ToGroup. Additional commands appear on lines 65 (Set-ADUser), 69 (Copy-DefaultDocs), and 75 (New-VirtualDir). Of these, only Set-ADUser is a cmdlet, one that lives in the

ActiveDirectory module. Do you see anywhere in this script where the ActiveDirectory module is imported? If not, then the module will need to be imported prior to running this script.

Numerous pieces of information will probably require customization, such as the UNC path assigned to \$homeunc on line 58, for example.

From a functionality perspective, this script appears to create a new user, add the user to a group, and create a home folder on a file server for the user. It also uses a command called Set-FolderACL to set permissions on that home folder. The home folder path is assigned to the user's account in Active Directory; this occurs on line 65. A set of default documents are copied to the home directory on line 69. Note that the use of clear command names, such as Copy-DefaultDocs, helps make this script's purpose easier to figure out.

Open E:\Mod10\democode\New-User.ps1. This is one of the six scripts that Provision.ps1 dot sourced. Here, we can see that this does define the New-User command as an *advanced function*. This type of function looks and works almost exactly like a cmdlet written in Visual Basic or C#; Appendix A contains more information about these cmdlets. A variety of parameters are defined. Do you see any hardcoded information that you might need to customize? One possibility is the default value for the \$InitialPassword parameter. Currently, that default value is Pa\$\$w0rd. Notice that because the \$ character normally has a special meaning to the shell, the backtick character is used to escape both \$ characters so that they are read as literals.

The \$upn variable, on line 65, may need customizing, too. It currently uses the contoso.com domain, which would not be appropriate for other environments.

In addition, the end of the script file, at line 92, contains two examples of how to use this function. This was a helpful feature added by the author that explains what this function does and how it is used.

## Discussion: Is This a Useful Tool?

- Would you include these tasks in a new user provisioning process in your organization?
- Are there other tasks you would include?
- What do you need to know before you can begin using this script?
  - What kind of input is required?
  - Is there any logging facility?
  - What permissions are needed to run it?
  - What other information would you need, or concerns would you have?



### Key Points

Would you use this type of script in your environment? What modifications, additions, or subtractions would you need to make in order for this to be the most useful in your environment?

What type of information would you need to know before you used this script?

As a group, review all seven script files and identify pieces of information that might need to be changed or customized in order for the script to function in a different environment. Also, make an inventory of the commands in each script. What commands are familiar to everyone in the group? Which commands will you need to learn more about in order to understand exactly what the script is doing?

## Lesson 2

# Understanding Scripts

- Review the example script with particular attention to selected scripting constructs



After completing a cursory examination of the seven scripts in the example, you will know and understand the script's major functionality. Now, you will examine the scripts in more detail to identify and learn about various scripting constructs present in Windows PowerShell.

### Lesson Objectives

After completing this lesson, you will be able to:

- Identify and describe key scripting constructs in Windows PowerShell.

MCT USE ONLY. STUDENT USE PROHIBITED

## Scripting Constructs

- We will review three main kinds of constructs:
  - Logical: These constructs enable the script to make a decision, such as whether or not a user should be added to a group
  - Loops: These constructs repeat some action
  - Functions: These constructs modularize a set of commands, creating a self-contained unit that can be reused more easily from other scripts (almost as if you were creating your own cmdlet)
- There are a few constructs beyond those that we will review here, and a list will be provided later

### Key Points

There are three main types of constructs in Windows PowerShell. While you can use these constructs from the command line, they are most frequently used in scripts that automate longer and more complex multi-step processes or tasks.

- **Logical constructs** are designed to make decisions, meaning they can execute different commands based on specific criteria or comparison.
- **Loops** are designed to repeat a given command, or a set of commands, for a specific number of times. Some loops can repeat commands until a certain condition is satisfied.
- **Functions** are designed to modularize one or more commands into a more easily-reused unit of work. For example, if you frequently run Invoke-Command with a number of parameters, such as port numbers or alternate credentials, then modularizing that as a function can make it easier to run the cmdlet with those exact parameters and much less typing.

## Demonstration

- We will review these scripts and specific portions of them, on-screen
- You have the scripts in your virtual machine image; you may wish to open the scripts in the Windows PowerShell ISE and follow along so that you can see the commands more clearly
- Do not attempt to run the script at this time

### Key Points

We will now examine the scripts in more detail, with a focus on specific scripting constructs. Keep in mind that you may encounter cmdlets or other commands that you have not used before; use the Help feature in Windows PowerShell to review the syntax for these cmdlets. It will help you understand what they do.

Begin by opening E:\Mod10\democode\Provision.ps1 in the Windows PowerShell ISE.

On line 5, note that this script accepts two input parameters, named \$file and \$log. The \$file parameter is required; if you do not provide it when running the script, then the script will display an error. The \$log parameter defaults to the filename "ADProvision.log."

These parameters are a form of *variable* in Windows PowerShell. Variables are identified by a \$ in front of the variable name; the variable name usually consists of letters, numbers, and the underscore character. Variable names like \${Hello There} are also permitted. In this case, the variable name contains a space, and the {brace} delimiters contain the entire variable name. The \$ is not part of the variable name; rather, it is a visual indicator that what *follows* is a variable name.

Lines 9-14 use dot sourcing to include the six task-specific scripts into the script. We will examine those scripts later in this course.

Line 16 begins a new *function*. A function is a scripting construct that enables one or more commands to be self-contained. Notice that this is a *parameterized function*; much like the script itself, it accepts two parameters named \$logfile and \$message. Neither parameter defines a default value.

Within the function, information is formatted and written to a text file. By including the two commands inside a function, the remainder of the script can more easily write messages to a log file. Each log message includes the current date as well as the logon user and domain, along with whatever message was passed in to the \$message parameter. The function is defined at the top of the script because, in Windows PowerShell, functions must be defined before they can be used. By defining the function first, the author ensures that the remainder of the script will have access to the function.

Notice that the function's commands are enclosed within curly braces. Also notice that the function's commands are indented. The indentation helps make it clear that these commands are enclosed within the function. Furthermore, user comments begin with a # character. These comments help explain what the commands within the function are doing.

Line 27 initiates an If construct. Immediately following the If keyword that's contained in parentheses, there's a comparison. You can put anything in the parentheses that you want, provided that the final result is either True or False. In this case, the Test-Path cmdlet determines if the filename provided to the \$file parameter exists or not. If the file exists, then Test-Path will return True. If it does so, then the conditional code that begins after the curly brace {} will execute.

Scroll down to line 83. This is the closing brace for the conditional code, meaning all of the commands from line 28 to line 82 will only execute if that file exists. Line 84 shows an optional component of the construct: The Else section will execute if the original condition was False. Therefore, if the file does not exist, then line 86 will execute. That line simply writes a message to the shell's Warning pipeline. Messages in that pipeline are generally displayed in an alternate color to call attention to them.

The last executable line of this script is line 90, which uses the Logit function that was defined earlier to write a "Finished" message to the log file.

Let's go back and examine the code that would have executed if the file did exist. This starts on line 29, where the Logit function logs a "starting" message to the log file.

We aren't going to review each line of the script at this point; we'll continue focusing on the major scripting constructs. You'll notice on line 43 that the ForEach-Object cmdlet is used (or rather, its alias, ForEach) at the end of the line. This isn't a scripting construct, although it does work similarly to one.

Now open E:\Mod10\democode\Set-FolderACL.ps1.

This script contains a function that begins on line 3. Lines 7 through 16 define the parameters for this function. The [CmdletBinding()] attribute on line 5 defines this as an *advanced function*, meaning it uses cmdlet-style named and positional parameters. Many attributes in the Param() section define the characteristics of the parameters, designating them as optional, as accepting pipeline input, and so on.

This type of function accepts one or more objects from the pipeline. A special PROCESS script block, on line 20, will execute one time for each object that was piped in. This behavior makes this function especially useful for performing a task against a series of objects. In this case, it sets permissions on home folders. Its input parameters define the folder path or paths; for each path piped into the function, the PROCESS script block will execute to set the permissions on each.

Now open E:\Mod10\democode\New-VirtualDir.ps1.

This script defines a simpler kind of function that, like the main Provision.ps1 script, defines input parameters. Three parameters are defined for this function. It uses the WebAdministration module, which is normally present on Windows Server® 2008 R2 computers that have the Web Server (IIS) role installed.

You'll notice the If construct used again, on line 9 and on line 13. The construct on line 9 actually ends on line 17. The author chose not to indent the code within the construct, and you can see that it makes it more difficult to determine where one construct ends.

You may want to indent lines 10 through 16 to improve the formatting of this script, which will make your review easier. The construct on lines 13 through 16 are nested within the first construct.

Pay close attention to line 13. The construct's comparison is simply a single variable, \$BrowseEnable. These constructs do not specifically need a comparison; however, they need True or False. Provided \$BrowseEnable will contain either True or False, then you don't have to explicitly compare it to the \$True

MCT USE ONLY. STUDENT USE PROHIBITED

or \$False values. In this case, \$BrowseEnable is one of the input parameters for this function, and it is defined as a [switch] type of variable. The [switch] type permits the use of True or False, ensuring that \$BrowseEnable will contain a value that the If construct can work with.

*As a group,* review the remaining scripts in this example. Can you identify other scripting constructs or things that look like scripting constructs?

The final module of this course presents all of the Windows PowerShell scripting constructs. In addition, you will discover reviews of scripting-specific concepts such as variables, arrays, scope, and topics such as debugging, error-handling, and various supported functions for modularization. Furthermore, the Appendix at the end provides information on related topics such as advanced cmdlets, comment-based Help information, and so on.

Your instructor may review some or all of that information in class, or assign one or more lessons for self-study.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 11

## Writing Your Own Windows PowerShell® Scripts

### Contents:

<b>Lesson 1:</b> Variables, Arrays, Escaping, and More Operators	11-3
<b>Lab A:</b> Using Variables and Arrays	11-30
<b>Lesson 2:</b> Scope	11-39
<b>Lesson 3:</b> Scripting Constructs	11-52
<b>Lab B:</b> Using Scripting Constructs	11-74
<b>Lesson 4:</b> Error Trapping and Handling	11-81
<b>Lab C:</b> Error Trapping and Handling	11-103
<b>Lesson 5:</b> Debugging Techniques	11-111
<b>Lab D:</b> Debugging a Script	11-125
<b>Lesson 6:</b> Modularization	11-130
<b>Lab E:</b> Modularization	11-153

## Module Overview

- Use variables, arrays, escape characters, and advanced operators
- Use all of the Windows PowerShell scripting constructs
- Describe Windows PowerShell's scope hierarchy
- Trap and handle run-time errors within a script
- Debug scripts using a variety of techniques and tools
- Modularize scripts by writing various types of functions



You have already learned that, at its simplest, a Windows PowerShell® script is a sequence of commands that you want executed in order. This type of script is similar to the batch files or batch scripts used by most other shells, including the Cmd.exe shell in Windows.

At a more complex level, Windows PowerShell scripts can begin to resemble a software program, such as those written in languages like Microsoft® Visual Basic® Scripting Edition (VBScript), JavaScript, JScript®, Perl, and so forth. Also, you want to create scripts that can anticipate and handle certain errors, and you may need to spend time debugging your scripts to get them working perfectly. You want to modularize your scripts, as well, so that you can more easily use portions of them in different scenarios.

You will learn to use these more complex scripting elements in this module.

Because most of the shell's functionality is provided by task-oriented cmdlets, its scripting language is actually quite simple, consisting of less than two dozen major keywords and constructs.

### Module Objectives

After completing this module, you will be able to:

- Use variables, arrays, escape characters, and advanced operators.
- Use all of the Windows PowerShell scripting constructs.
- Describe the scope hierarchy of Windows PowerShell.
- Trap and handle run-time errors within a script.
- Debug scripts using a variety of techniques and tools.
- Modularize scripts by writing various types of functions.

## Lesson 1

# Variables, Arrays, Escaping, and More Operators

- Explain the use of variables
- Explain variable naming conventions
- Create, modify, and remove variables using cmdlets, ad-hoc syntax, and the variable: drive
- Perform simple mathematical calculations using operators
- Explain the difference between single and double quotation marks in the shell
- Compare objects using –contains, -is, and –like
- Create an array of values
- Access individual elements in an array or collection
- Use the escape character in the shell
- Create and use hashtables



You have already seen variables used in Windows PowerShell in earlier modules, but in this lesson you will learn more about what they are, how they work, and how you use them within the shell. You will also learn about related topics, such as arrays, and additional operators.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the use of variables.
- Explain variable naming conventions.
- Create, modify, and remove variables using cmdlets, ad-hoc syntax, and the variable: drive.
- Perform simple mathematical calculations using operators.
- Explain the difference between single and double quotation marks in the shell.
- Compare objects using –contains, -is, and –like.
- Create an array of values.
- Access individual elements in an array or collection.
- Use the escape character in the shell.
- Create and use hashtables.

MCT USE ONLY. STUDENT USE PROHIBITED

## Variables

- Think about shortcuts in Windows Explorer
  - They have properties, such as the working directory that will be set when the shortcut is opened, and the path of the file that the shortcut points to
  - They can be manipulated: You can move them, delete them, copy them, and so forth
  - They are not usually of primary interest. Rather, shortcuts *point* to something of primary interest, such as an application or a document
  - They point to something that is physically stored in a different location
- Shortcuts in Windows Explorer are a lot like variables in Windows PowerShell
  - Although a variable is an object in and of itself, you are usually more interested in what the variable points to

**Get-Command -noun variable**

### Key Points

Take a moment to think about shortcuts in Windows Explorer. You have doubtless created shortcuts to files and folders, and the Start menu consists of little more than sets of folders and shortcuts. Shortcuts have a few relevant attributes:

- They have properties, such as the working directory that is set when the shortcut is opened and the path of the file that the shortcut points to.
- They can be manipulated: You can move them, delete them, copy them, and so forth.
- They are not usually of primary interest. Rather, shortcuts *point* to something of primary interest, such as an application or a document.
- They point to something that is physically stored in a different location.

In Windows PowerShell, variables serve a purpose similar to shortcuts. Although a variable is an object in and of itself, you are usually more interested in what the variable points to. Variables act as a sort of named shortcut to something that the shell is storing in memory, such as a string, or a number, or any kind of object—or even a collection of objects.

Like most of the dynamically-created elements in the shell, variables do not persist once the shell is closed. Windows PowerShell provides a variable: `drive`, which stores all of the variables that have been defined in the shell. The variable: `drive` contains not only the variables you create, but also a large set of variables that are defined by the shell and used to hold configuration settings that affect the shell's behavior.

The shell provides numerous cmdlets for managing variables; you can see them all by running this command:

**Get-Command -noun variable**

However, in many cases you do not use these cmdlets. The shell also provides an ad-hoc syntax that allows you to define and modify variables without the use of cmdlets. You can also manage variables by using the variable: `drive`; for example, you can delete an item from that drive to remove that variable from memory.

MCT USE ONLY. STUDENT USE PROHIBITED

## Variable Naming

- Variable names usually consist of letters, numbers, and underscores, but not the '\$' character
  - The special '\$' character is used to cue the shell that what follows is a variable name
  - When you pass a variable name to a cmdlet parameter that expects a variable name, omit the '\$' character
  - When you use the '\$' character, PowerShell automatically passes the **contents** of the variable rather than the name

```
$var = 'computername'
```

- Variable names can contain spaces; however using spaces requires variables to be wrapped in curly braces

```
 ${My Variable} = 5
```

- Avoid using spaces when possible

### Key Points

Variable names usually consist of letters, numbers, and underscores. A variable's name *does not* include a \$ character; the \$ character is used as a cue to tell the shell that what follows is a variable name. When you are passing a variable name to a cmdlet parameter that expects a variable name, you do not include the \$ character:

```
Remove-Variable -name var
```

If you do include a \$ in front of the variable name, the shell passes the *contents* of the variable to the parameter:

```
$var = 'computername'  
Remove-Variable -name $var
```

This example will remove the variable \$computername, not the variable \$var.

Variable names can contain spaces and other characters, but doing so requires that you wrap the variable name in curly braces:

```
 ${My Variable} = 5
```

Using spaces is generally considered to be a poor practice, since the curly braces require more typing and make your scripts somewhat more difficult to read.

## Assigning Objects to a Variable

- Variables in PowerShell can contain more than simple text. A PowerShell variable can also contain an object or a collection of objects.
- This object assignment makes PowerShell variables very powerful in your scripting.
  - The following example assigns the text value of '5' to the variable '\$var'.

```
$var = 5
```

- The following example assigns the object bound by 'Get-Service' to the variable '\$services'.

```
$services = Get-Service
```

### Key Points

Windows PowerShell uses the equal sign (=) as an assignment operator. When used, whatever is on the right side of the operator is assigned to whatever is on the left side:

```
$var = 5
```

The shell permits you to create new variables by assigning a value to the variable, exactly as in this example. You do not need to declare or define variables in advance before assigning something to them.

**Note:** If you are familiar with VBScript, you may wonder if there is an equivalent to Option Explicit. That will be addressed in the next lesson.

Any object or collection of objects can be assigned to a variable:

```
$services = Get-Service
```

A variable retains its contents until you change the contents, until you close the shell, or until you remove the variable.

## Managing Variables

- Variables are created, used, and managed through multiple techniques in the shell

```
$var = "Hello"  
$var = 100  
$var = ''  
$var = $null
```

- Six cmdlets are also available for managing variables
  - New-Variable      Set-Variable      Clear-Variable
  - Remove-Variable    Get-Variable      Dir variable:
- Variables can be piped to other cmdlets. When piping, it is the variable's contents that are piped
- Techniques exist to view and work with variable contents

```
$wmi = Get-WmiObject Win32_OperatingSystem  
$wmi.Caption  
$wmi.Reboot()
```

### Key Points

You can create and use variables ad-hoc in the shell, by referring to the variable:

```
$var = 100  
$var  
$var = "Hello"
```

Note that clearing a variable is not the same as removing it. In the following example, the variable \$var still exists, but it contains a value of null:

```
$var = 100  
$var = ''  
$var = $null
```

You can also create, set, and remove variables by using cmdlets:

```
New-Variable -name var -value (Get-Process)  
Set-Variable -name var -value $null  
Clear-Variable -name var  
Remove-Variable -name var
```

The last command in that example actually removes the variable. You can confirm this by running:

```
Get-Variable
```

Or by examining the contents of the variable drive:

```
Dir variable:
```

Variables can be piped to other cmdlets; when you do so, it is actually the variable's *contents* that are piped:

```
$services = Get-Service  
$services | Sort Status | Select -first 10
```

Finally, a variable enables you to access the members—the properties and methods—of an object. Treat the variable as if it were the object: Follow its name by a period and then the name of the member you want to access:

```
$wmi = Get-WmiObject Win32_OperatingSystem  
$wmi.Caption  
$wmi.Reboot()
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Managing Variables

- Learn how to manage variables by using a variety of techniques



### Key Points

In this demonstration, you will learn how to manage variables by using a variety of techniques.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Managing Variables.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Quotation Marks

- Single and double quotation marks can be used to delimit strings of characters.

- The following two statements are processed identically.

```
$var = "Hello"  
$var = 'Hello'
```

- When using double quotation marks, the shell looks for the '\$' character and assumes what follows is a variable.

- When executed, the shell will replace the variable name with its contents.

<b>\$var2 = "\$var World"</b>	->	<b>Hello World</b>
<b>\$var3 = '\$var World'</b>	->	<b>\$var World</b>

- It is a common practice to only use double quotation marks only when you need variable replacement. Otherwise, use single quotation marks.

### Key Points

The shell permits you to use either single or double quotation marks to delimit strings of characters. For the most part, either type of quotation mark can be used in pairs. The following statements are functionally identical:

```
$var = "Hello"  
$var = 'World'
```

The key difference between them is that, within *double* quotation marks, the shell looks for the \$ character. When it finds that character, it assumes that the following characters, to the next white space, are a variable name, and it replaces the entire variable with its contents. The following examples illustrate the difference:

```
PS C:\> $var = 'Hello'  
PS C:\> $var2 = "$var World"  
PS C:\> $var3 = '$var World'  
PS C:\> $var2  
Hello World  
PS C:\> $var3  
$var World
```

It is a common practice to use double quotation marks only when you specifically need the variable-replacement feature and to use single quotation marks otherwise.

## Demonstration: Quotes in the Shell

- See the difference between single and double quotation marks



### Key Points

In this demonstration, you will see the difference between single and double quotation marks.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Quotes in the Shell.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Escape Character

- PowerShell's escape character `` is used to take away the special meaning of the immediately-following character
  - Conversion of a space to a literal character. This is useful for path names that contain spaces
  - Conversion of a dollar sign to a literal dollar sign. This is useful when the dollar sign is needed as text
- The escape character can also remove the special meaning from the carriage return
  - This process can be used to split a single-line PowerShell command into multiple lines for better readability
- It can further be used to add special meaning to certain otherwise nonspecial characters
  - ``t' creates a tab. ``n' creates a new line. ``r' a return

**help about\_escape\_characters**

### Key Points

Like many shells, Windows PowerShell has an escape character. Its escape character is the backtick, or ` character. On US keyboards, this character is commonly located on the upper-leftmost key below or near the Esc or Escape key, usually on the same key as the tilde (~) character but this will vary for international keyboards in different regions of the globe. See the reference below for details on various international keyboards.

**Note:** In some fonts, it can be difficult to distinguish between the backtick ` and a single quotation mark or apostrophe '. Be careful not to confuse the two.

The escape character takes away any special meaning of the character following it. For example, in Windows PowerShell, the space character has a special meaning and is used as a delimiter. When escaped, however, that special meaning is taken away and the space becomes a literal character:

```
Cd c:\program` files
```

You have already learned that the \$ character has special meaning, and that the shell looks for it within double quotation marks for the variable replacement feature. Take away that special meaning, however, and you are left with a dollar sign:

```
PS C:\> $var = 'Hello'  
PS C:\> $var2 = "$var World"  
PS C:\> $var3 = ``$var contains $var"  
PS C:\> $var2  
Hello World  
PS C:\> $var3  
$var contains Hello
```

The escape character can even take away the special meaning of a carriage return, which is normally used to delimit logical lines:

```
$var = "This `n is a `r single line"
```

However, this practice is usually unnecessary because the shell automatically knows how to parse this kind of multiple-line statement correctly.

Finally, the escape character can add special meaning to certain characters that usually do not have a special meaning. The `t sequence stands for a tab, `n for a newline, `r for a carriage return, and so forth; run help about\_escape\_characters for full documentation and other special escape sequences.

## Reference

- Windows International Keyboard Layouts,  
<http://go.microsoft.com/fwlink/?LinkId=200517>.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Escape Characters

- See how to use escape character sequences



### Key Points

In this demonstration, you will see how to use escape character sequences.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Escape Characters.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Arrays

- Arrays are variables that contain more than one object
  - Assigning variables to objects, as opposed to text, can place arrays of objects into the variable

```
$var = Get-Service
```
  - Comma-separated lists of text are automatically converted to arrays by the shell

```
$myarray = 1,2,3,4,5,6,7,8,9,0
```
  - Index numbers are used to refer to individual elements, with the first object in the array having an index of 0

```
$services = Get-Service  
$services[0]  
$services[1]
```
  - Arrays also have a Count property, which exposes the number of elements in the array

```
$services.Count
```

### Key Points

An array is a variable that contains more than one object. For example:

```
$var = 5
```

In this example, \$var is not an array—it contains only a single object, the integer 5. However, this example places an array of objects into the variable:

```
$var = Get-Service
```

In the world of software development, there is a distinct difference between an array of values and a collection of objects. Windows PowerShell abstracts many of these differences, and so in many cases the terms *array* and *collection* can be used to mean the same thing.

The shell automatically treats any comma-separated list as an array:

```
$myarray = 1,2,3,4,5,6,7,8,9,0
```

There is also a more formal way of declaring a new variable: Using the @ operator, and enclosing the list of values in parentheses:

```
$myarray = @(1,2,3,4,5,6,7,8,9,0)
```

There is no functional difference between these two examples of creating an array.

When using an array, you can use index numbers to refer to individual elements. The first object in the array has the index 0. Using negative indexes enables you to access the elements from the end of the array: -1 is the last item, -2 is the next-to-last item, and so on:

```
$services = Get-Service  
$services[0]
```

```
$services[1]  
$services[-1]
```

Arrays also have a Count property that exposes the number of elements in the array:

```
$services.Count
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Arrays

- See how to use arrays



### Key Points

In this demonstration, you will see how to use arrays.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Working with Arrays.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Hashtables

- Hashtables are special kinds of arrays where each element consists of a key-value pair.
  - Each key contains a value. Each value can be referenced by dot-referencing its key.

```
PS C:\> $hash = @{"Server1"="192.168.15.4";  
"Server2"="192.168.15.11";"Server3"="192.168.15.26"}  
PS C:\> $hash.Server1  
192.168.15.4
```

- Hashtables also have Count properties.
- Hashtables can be piped to Get-Member to see other properties and methods.

**\$hash | gm**

### Key Points

You have already seen how to use a hashtable with cmdlets such as Select-Object and Format-Table. A *hashtable* is a special kind of array. Each element within the array consists of a key and a value, called a *key-value pair*. When you used a hashtable with Select-Object, for example, you provided two elements: The first element's key was Label, and the second was Expression:

```
Get-Process | Select @{{Label='TotalMem';Expression={$_.VM + $_.PM}}}
```

The @ operator, combines with curly braces, is used to create hashtables. When creating your own hashtable, you can select the keys and values yourself; you are not restricted to Label and Expression. After you have created the hashtable, you can use the keys to retrieve specific values:

```
PS C:\> $hash = @{"Server1"="192.168.15.4";"Server2"="192.168.15.11";  
"Server3"="192.168.15.26"}  
PS C:\> $hash.Server1  
192.168.15.4
```

**Note:** The @ is used as an operator to create both arrays and hashtables (which are also called *associative arrays*); simple arrays delimit values within parentheses, whereas hashtables delimit the key-value pairs in curly braces. Pay close attention to this distinction.

A hashtable, like any array, has a Count property:

```
$hash.Count
```

You can pipe a hashtable object to Get-Member to see other properties and methods:

```
$hash | gm
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Hashtables

- Learn how to use hashtables



### Key Points

In this demonstration, you will learn how to use hashtables.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Hashtables.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Splatting

- Splatting is a way to enter parameter names and values into a hashtable and then passing the entire hashtable to a cmdlet
  - This is useful for dynamically constructing parameters to pass to a cmdlet
- First, create a normal hashtable where the keys are parameter names and the values are the values for those parameters

```
$parms = @{"ComputerName"="Server-R2";
    "Class"="Win32_BIOS";
    "Namespace"="root\cimv2"
}
```

- Then, use the '@' operator and the variable name (without the '\$' character) to pass the hashtable to a cmdlet

```
Get-WmiObject @parms
```

### Key Points

*Splatting* is a way of putting parameter names and values into a hashtable, and then passing the entire hashtable to a cmdlet. This can be useful for dynamically constructing parameters to pass to a cmdlet. To begin, create a normal hashtable where the keys are parameter names, and the values are the values for those parameters:

```
$parms = @{"ComputerName"="Server-R2";
    "Class"="Win32_BIOS";
    "Namespace"="root\cimv2"
}
```

Then, use @ as a splat operator and the variable name—which *does not include the \$ character*—to pass the hashtable to a cmdlet:

```
Get-WmiObject @parms
```

The cmdlet runs normally, taking the parameter names and values from the hashtable.

**Note:** These differing uses for the @ symbol can be confusing. Its exact purpose depends on the context in which it is being used, and all of the uses relate in some way to arrays or hashtables (associative arrays).

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Splatting

- Learn how to use splatting in the shell



### Key Points

In this demonstration, you will learn how to use splatting in the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Splatting.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Arithmetic Operators

- Arithmetic operators can perform mathematical calculations in the shell and are evaluated by the shell in standard order-of-operation rules
  - Multiplication (\*) and division (/) from left to right
  - Addition (+) and subtraction (-) from left to right
  - Parenthetical expressions are evaluated from left to right and from the most deeply-nested expression outward
- The shell recognizes common Base-2 values
  - KB, MB, GB, TB. KB as 1,024, and so on
- The '+' character can also be used for string concatenation

```
PS C:\> $var1 = "One "
PS C:\> $var2 = "Two"
PS C:\> $var3 = $var1 + $var2
PS C:\> $var3
One Two
```

### Key Points

Arithmetic operators are used to perform mathematical calculations in the shell. These are evaluated by the shell according to standard order-of-precedence rules:

- Multiplication (\*) and division (/) from left to right.
- Addition (+) and subtraction (-) from left to right.
- Parenthetical expressions are evaluated from left to right, and from the most deeply-nested expression outward.

The shell recognizes KB, MB, GB, and TB as sizes. KB is equal to 1,024, MB is equal to 1,048,576, and so forth. Note that these are base-2 values, not base-10 values (meaning 1 KB is not exactly equal to 1,000).

Note that + is also an operator for string concatenation:

```
$var1 = "One "
$var2 = "Two"
$var3 = $var1 + $var2
$var3
One Two
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Math in the Shell

- Learn how to use the mathematical operators



### Key Points

In this demonstration, you will learn how to use the mathematical operators.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Math in the Shell.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Variable Types

- The shell I permit you to put any type of object into any variable. If an operation requires a specific type, the shell attempts to convert (or, coerce) objects into the necessary type.
- Common types:
  - String, Int (integer), Single and Double (floating integers), Boolean, Array, Hashtable, XML, Char (a single character)
- Type coercion can have unexpected results, when coercion inappropriately changes types to fit the operation.
  - \$a is integer, \$b is string -> \$a + \$b = 10 -> ????
- One solution is to explicitly inform the shell of the variable's type. This is done using a .NET Framework type name in square brackets during variable assignment.

```
[int]$a = "5"  
[string]$a = "Hello"
```

### Key Points

The shell normally permits you to put any kind of object into any variable. If you perform an operation that requires a specific type of object, the shell tries to temporarily convert, or *coerce*, objects into the necessary type. For example:

```
PS C:\> $a = 5  
PS C:\> $b = "5"  
PS C:\> $a + $b  
10  
PS C:\> $b + $a  
55
```

The variable \$a is an integer, but the variable \$b is a string—because its value was delimited in quotation marks. In the example of \$a + \$b, the shell produced 10. The object in \$b was temporarily coerced to be an integer to complete the mathematical operation. The shell decided to do so because the first variable contained an integer, and the second variable contained an object that could be perceived as an integer.

In the example of \$b + \$a, the opposite occurred: The shell saw a string first, and so it coerced the second variable to be of a compatible type for string concatenation.

This type of behavior can sometimes be confusing and lead to unexpected results. One solution is to explicitly inform the shell what object type for which you plan to use a variable. You do this by specifying a .NET Framework type name in square brackets:

```
PS C:\> [int]$a = "5"  
PS C:\> $a | gm  
TypeName: System.Int32
```

In this example, even though "5" is a string, the shell was forced to permanently convert it to an integer to place it into the variable \$a. If the conversion had not been possible, the shell would have displayed an error:

```
PS C:\> $a = "Hello"
Cannot convert value "Hello" to type "System.Int32". Error: "Input string was not in a
correct format."
```

After you explicitly provide a type for a variable, the shell respects that type until you explicitly change it:

```
PS C:\> [string]$a = "Hello"
```

Some of the common types are:

- String
- Int (integer)
- Single and Double (floating numbers)
- Boolean
- Array
- Hashtable
- XML
- Char (a single character)

Particularly when writing scripts, it is considered a best practice to explicitly type all variables upon first use. Doing so helps to prevent certain kinds of errors or unexpected behaviors and can generally make debugging easier.

## More Comparison Operators

- You've already seen the basic comparison operators, such as `-eq` and `-gt`
- Other advanced comparison operators exist
  - `-contains` tells you if a particular object is within a collection
  - `-like` performs case-insensitive wildcard string comparisons
  - `-clike` performs case-sensitive wildcard string comparisons
  - `-notlike` and `-cnotlike` are the logical reverse of `-like` and `-clike`
  - `-is` tells you if an object is of a particular type or not
  - `-as` converts a given object to another type

### Key Points

You have already learned about basic comparison operators such as `-eq` and `-gt`; the shell provides additional comparison operators.

`-contains` tells you if a particular object is contained within a collection. This does *not* perform simple string matches; for example, it does not tell you whether the string "Server1" contains the substring "ver" or not. Rather, it matches entire objects:

```
$collection = "One", "Two", "Three"  
$collection -contains "One"
```

The `-contains` operator matches more complex objects by examining all of the objects' properties. To reverse the logic and see if a collection does not contain an object, use `-notcontains`.

The `-like` operator performs wildcard string comparisons, and is normally case-insensitive.

```
$var = "Server1"  
$var -like "server*"
```

A case-sensitive version, `-clike`, is available. To reverse the logic, use `-notlike` and `-cnotlike`. All four operators use `*` as a wildcard character.

The `-is` operator tells you if an object is of a particular type or not.

```
$var = "CONTOSO"  
$var -is [string]
```

Closely related is the `-as` operator, which attempts to convert a given object to another type. If a conversion is not possible, it generally produces an empty value, such as a blank string or a zero.

```
PS C:\> $var = "CONTOSO"
```

```
PS C:\> $var2 = $var -as [int]
PS C:\> $var2
PS C:\>
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: More Comparison Operators

- Learn how to use additional comparison operators



### Key Points

In this demonstration, you will learn how to use additional comparison operators.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\More Comparison Operators.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab A: Using Variables and Arrays

- Exercise 1: Creating Variables and Interacting with Them
- Exercise 2: Understanding Arrays and Hashtables
- Exercise 3: Using Single-Quoted and Double-Quoted Strings and the Backtick
- Exercise 4: Using Arrays and Array Lists
- Exercise 5: Using Contains, Like, and Equals Operators

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You've been using PowerShell interactively for long enough that you're comfortable with the cmdlets you use. Now you want to start automating your environment and you need to create scripts that work on a larger scale. Your developer friend told you about variables, arrays, and hashtables. He also suggested that you need to learn the key looping constructs like for, while, do, etc., so that you can take advantage of them in your automation scripts.

#### Lab Setup

Before you begin the lab you must:

1. Start the **LON-DC1** and **LON-SVR1** virtual machines and log on with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. On LON-SVR1, open the Windows PowerShell ISE. All PowerShell commands are run within the Windows PowerShell ISE program.

## Exercise 1: Creating Variables and Interacting with Them

### Scenario

You've been using PowerShell interactively for long enough that you're comfortable with the cmdlets you use. You have noticed that you often repeat the same command when working interactively and would like to learn how to avoid unnecessary repetition. A colleague told you that you can store results in variables so you need to experiment with variables and learn how they work.

The main tasks for this exercise are carried out on LON-SVR1 and are as follows:

1. Create strongly- and weakly-typed variables.
2. View all variables and their values.
3. Assign values to variables from a child scope and learn how scopes affect variables.
4. Remove a variable.

#### ► Task 1: Create and work with variables and variable types

1. Create a variable called **myValue** and assign the value of **1** to it.
2. Assign "Hello" to the value of **myValue**. Notice that the variable changed from an integer to a string.
3. Read help documentation and examples for the **Get-Variable** cmdlet.
4. Show all properties for the **myValue** variable using **Get-Variable**.
5. Show the contents of the **myValue** variable using **Get-Variable** with the **ValueOnly** parameter.
6. Redefine the variable **myValue** assigning it as an **[int]** (integer). Assign the value of **1** to it.
7. Try to assign the value of "Hello" to the **myValue** variable. Notice that an error was raised in PowerShell because the variable **myValue** has been typed as an **[int]**.

#### ► Task 2: View all variables and their values

- Get a list of all variables and their current values in the current scope by using the **Get-Variable** cmdlet.

#### ► Task 3: Remove a variable

1. Read the help documentation and examples for the **Remove-Variable** cmdlet to learn how you can use it to remove variables.
2. Remove the **myValue** variable you created earlier using the **Remove-Variable** cmdlet.

**Results:** After this exercise, you should be able to create new variables, view variable values and remove variables.

## Exercise 2: Understanding Arrays and Hashtables

### Scenario

When you are automating tasks in your environment, there are times when you would like to store a collection of objects in a variable and work with that collection. You want to explore how to do this using arrays and hashtables.

The main tasks for this exercise are as follows:

1. Create an array of services.
2. Access individual items and groups of items in an array.
3. Retrieve the count of objects in an array.
4. Create a hashtable of services.
5. Access individual items in a hashtable.
6. Enumerate key and value collections of a hashtable.
7. Remove a value from a hashtable.

► **Task 1: Create an array of services**

1. Read the documentation in **about\_Arrays** to understand how arrays work in PowerShell.
2. Create a variable called **services** and assign it to a list of all Windows services.
3. Show the contents of the **services** variable.
4. Show the type of the **services** variable.

► **Task 2: Access individual items and groups of items in an array**

1. Show the first item in the services array.
2. Show the second item in the services array.
3. Show the last item in the services array.
4. Show the first ten items in the services array.

► **Task 3: Retrieve the count of objects in an array**

- Determine how many items are in an array by using the **Count** property.

► **Task 4: Create a hashtable of services**

1. Read the documentation in **about\_Hash\_Tables** to understand how hashtables work in PowerShell.
2. Create a new, empty hashtable called **serviceTable**.
3. Use the services array to populate the **serviceTable** hashtable using the service name as the key and the service as the value.

**Hint:** One way to do this is to use the `ForEach` scripting construct, which you will learn about next. This is different from the `ForEach-Object` cmdlet. For example, if `$services` contains your services, and `$serviceTable` is your empty hashtable:

```
ForEach ($service in $services) {  
    $serviceTable[$service.Name] = $service  
}
```

This creates a hashtable `$serviceTable`, where the keys are service names, and the values are the actual service objects. You can access a service by its name.

4. Show the **serviceTable** hashtable contents.

► **Task 5: Access individual items in a hashtable**

1. Show the **BITS** entry in the **serviceTable** hashtable.

**Hint:** If `$serviceTable` is your hashtable, `$serviceTable['service_name']` accesses the service named `service_name`.

2. Show the **wuauserv** entry in the **serviceTable** hashtable.
3. Use **Get-Member** to show all properties and methods on the **serviceTable** hashtable.
4. Determine how many items are in the **serviceTable** hashtable.

**Hint:** Hashtables are objects, and one of their properties is `Count`. If you start with the variable that contains your hashtable, follow it with a period, and follow that with the `Count` property, you can retrieve the contents of the `Count` property.

► **Task 6: Enumerate key and value collections of a hashtable**

**Hint:** Hashtables are objects, and expose properties such as `Keys` and `Values`. These properties are *collections*, meaning they contain other objects - such as all the keys in the hashtable, or all the values in the hashtable, respectively. Use the `Keys` and `Values` properties in the same way you used the `Count` property earlier.

1. Show all keys in the **serviceTable** hashtable.
2. Show all values in the **serviceTable** hashtable.

► **Task 7: Remove a value from a hashtable**

1. Remove the **BITS** entry from the **serviceTable** hashtable using the appropriate method.

**Hint:** Hashtables have a `Remove()` method that can remove items identified by their key. For example, `$hash.Remove('Hello')` would remove the element having the key "Hello" from the hashtable in `$hash`.

2. Show the **BITS** entry in the **serviceTable** hashtable to verify that it was properly removed.

**Results:** After this exercise, you should be able to create arrays and hashtables, access individual items in arrays and hashtables, and remove items from hashtables.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Using Single-Quoted and Double-Quoted Strings and the Backtick

### Scenario

While automating tasks, you would like to format some of the output using strings. You need to understand how different strings work in PowerShell so that you can create compound strings that contain variable values. Also, while using PowerShell interactively you noticed that there are times when you would like to split a command across multiple lines.

The main tasks for this exercise are as follows:

1. Learn the differences between single-quoted and double-quoted strings.
  2. Use a backtick (`) to escape characters in a string.
  3. Use a backtick (`) as a line continuance character in a command.
- **Task 1: Learn the differences between single-quoted and double-quoted strings**
1. Create a variable called **myValue** and assign it a value of 'Hello'.
  2. Show the value of the following string by entering it into PowerShell: "The value is \$myValue". Note how the string variable was expanded in the double-quoted string.
  3. Show the value of the following string by entering it into PowerShell: 'The value is not \$myValue'. Note how the string variable was not expanded in the single-quoted string.
- **Task 2: Use a backtick (`) to escape characters in a string**
1. Show the value of the following string by entering it into PowerShell: "The value is not `\\$myValue". Make sure you enter the backtick before **\$myValue** and note how the string variable was not expanded in the double-quoted string when the \$ was escaped with a backtick.
  2. List the contents of the C:\Program Files directory.
  3. Repeat the last command without using quotation marks by using a backtick to escape the space between *Program* and *Files*.
- **Task 3: Use a backtick (`) as a line continuance character in a command**
- Retrieve the Windows Update service object using **Get-Service**. Place the **Name** parameter on the second line of the command and use a backtick at the end of the first line to indicate that the command spans two lines.

**Results:** After this exercise, you should be able to create simple and compound strings and use a backtick to split a single command across multiple lines.

## Exercise 4: Using Arrays and Array Lists

### Scenario

While working with arrays, you noticed that you were not able to easily find items in the array or remove items from the array. You need to learn about Array Lists so that you can manipulate the contents of the collection more easily.

The main tasks for this exercise are as follows:

1. Learn how to extract items from an array.
2. Discover the properties and methods for an array.

► **Task 1: Learn how to extract items from an array**

1. Create a variable called **days** and assign an array of the seven days of the week to that variable, starting with **Sunday**.
2. Show the third item in the **days** array.

► **Task 2: Discover the properties and methods for an array**

1. Pass the **days** array to **Get-Member** to see the properties and methods. Note that the properties and methods that appear are for the items in the **days** array and not for the **days** array itself.
2. Use **Get-Member** with the **InputObject** parameter to show the properties and methods for the **days** array. This technique also shows the properties and methods for the **days** array itself.

**Results:** After this exercise, you should be able to extract specific items from arrays by index or from array lists by name. You should also be able to discover the properties and methods for an array or array list.

## Exercise 5: Using Contains, Like and Equals Operators

### Scenario

Sometimes when you are working with your scripts, you find yourself working with collections of items. You need to explore how to find items in those collections that match or partially match a string.

The main tasks for this exercise are as follows:

1. Use contains, like and equals operators to find elements in arrays.
2. Understand the difference when using those operators with strings.

#### ► Task 1: Use contains, like and equals operators to find elements in arrays

1. Read the **about\_Comparison\_Operators** documentation that describes the **contains**, **like**, and **equals** operators to understand how they are used in PowerShell.
2. Create a new variable called **colors** and assign an array to it containing the following values: **Red**, **Orange**, **Yellow**, **Green**, **Black**, **Blue**, **Gray**, **Purple**, **Brown**, and **Blue**.
3. Use the **like** operator to show all colors in the colors array that end with **e**.
4. Use the **like** operator to show all colors in the colors array that start with **b**.
5. Use the **like** operator to show all colors in the colors array that contain **a**.
6. Use the **contains** operator to determine if the colors array contains the value **White**.
7. Use the **contains** operator to determine if the colors array contains the value **Green**.
8. Use the **equals** operator to find all values in the colors array that are equal to the value **Blue**.

#### ► Task 2: Understand the difference when using those operators with strings

1. Create a new variable called **favoriteColor** and assign the value of **Green** to it.
2. Use the **like** operator to determine whether the value in **favoriteColor** contains **e**.
3. Use the **contains** operator to determine whether the value in **favoriteColor** contains **e**. Note that this returns false because it treats the **favoriteColor** variable like an array of strings.

**Results:** After this exercise, you should be able use the contains, like, and equal operators to show partial contents of a collection. You should also understand how these operators function differently, depending on the type of object they are being used with.

## Lab Review

1. How do you retrieve the members (properties and methods) for a collection?
2. When should you use a double-quoted string instead of a single-quoted string?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 2

# Scope

- Explain the scope hierarchy in Windows PowerShell
- Read and modify local and out-of-scope variables



Like many programming and scripting languages, Windows PowerShell supports a *scope hierarchy*. This hierarchy controls when and where certain elements are visible and usable, and controls how those elements may be referenced and changed. Because scope is always operating, even if you are not aware of it, using the shell without a proper understanding of scope can create confusion and result in unexpected behaviors.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the scope hierarchy in Windows PowerShell.
- Read and modify local and out-of-scope variables.

MCT USE ONLY. STUDENT USE PROHIBITED

## Scope

- Scope is designed to provide boundaries around executable elements within the shell
  - It intends to keep different elements from improperly interacting with each other
  - It helps to make elements more self-contained
  - Variables, functions, aliases, and PSDrives are scoped
- The top-level scope is called the *Global Scope*
  - New scopes are created when you run a script or execute a function
  - Run a script
    - That executes a function
      - That executes another script
- When a script finishes executing, its scope is destroyed

### Key Points

Scope is designed to provide boundaries and borders around executable elements within the shell. When used properly, it helps keep different elements from interacting with one another improperly, and helps to make elements more self-contained.

The top-level scope in the shell is called the global scope. When you are working directly at the command line, and not in a script, you are working in the global scope. This scope contains certain predefined aliases, PSDrive mappings, variables, and even predefined functions.

All other scopes are children (or grandchildren, and so on) of the global scope. A new scope is created whenever you run a script or whenever you execute a function. This can result in a deeply nested hierarchy. For example, suppose that you:

- Run a script
  - That contains a function
    - That executes another script

That last script is the fourth scope created: The global scope, the scope for the first script, the scope for the function, and the scope for the second script.

Scopes last only as long as needed; when a script finishes executing, its scope is *destroyed* or removed from memory. Certain actions done within that scope automatically disappear after the scope is destroyed.

The following shell elements are scoped:

- Variables
- Functions
- Aliases

- PSDrives

MCT USE ONLY. STUDENT USE PROHIBITED

## Scope Rules

- Three rules for scopes
  - When an element is accessed, the shell looks to see if that element exists within the current scope. If it does, then the shell uses that element.
  - If an element does not exist in the current scope, the shell goes to the parent scope to see if the element exists there.
  - The shell continues to go up the scope hierarchy until it locates the element. If the shell makes it all the way to the global scope and doesn't find the element, then the element doesn't exist.

Global Scope: \$var = 100

Script Scope: \$var = 200

## Key Points

The shell enforces several default rules for scopes.

- When an element is accessed, the shell looks to see if that element exists within the current scope. If it does, then the shell uses that element.
- If an element does not exist in the current scope, the shell goes to the parent scope to see if the element exists there.
- The shell continues to go up the scope hierarchy until it locates the element. If the shell makes it all the way to the global scope and doesn't find the element, then the element doesn't exist.

For example, suppose you run a script that contains just one command:

```
Gwmi win32_service
```

Every scope starts out as a blank, fresh session. You haven't defined the alias "Gwmi" in this scope, and so when you use it, the shell goes up to the parent—the global scope—to see if the alias exists there. It does, by default, and so the shell uses the Gwmi alias that points to Get-WmiObject. This scope behavior is why all of the default aliases, PSDrives, and variables work within your scripts: They are defined in the global scope.

The behavior is different when you change an element. Normally, the shell does not allow you to change elements in a parent scope from within a child scope. Instead, elements are created within the current scope.

For example, suppose you run the following command from the command line—that is, in the global scope:

```
$var = 100
```

You now have a variable \$var defined within the global scope, and it contains the integer 100. The scope hierarchy looks like this:

- Global: \$var = 100

Next, you run a script that contains the following:

```
Write $var
```

This scope does not contain a variable \$var, and so the shell looks at the parent scope—which is the global scope. The variable \$var is defined there, and so the integer 100 is returned to the script, and the script outputs 100.

Now suppose you run a second script, which contains the following:

```
$var = 200  
write $var
```

The variable \$var is now created in the script's scope, and the integer 200 is placed into it. The scope hierarchy now looks like this:

- Global: \$var = 100
- Script: \$var = 200

A new variable named \$var has been created in the script scope, but the global scope's variable is unchanged. The script outputs 200 because that is what the variable \$var contains. This does not affect the global scope. If, after the script completes, you run this command from the command line:

```
Write $var
```

the output is 100, which is what \$var contains within the global scope.

This behavior applies to anything created within a child scope. The basic rules to remember are:

- If you *read* something that does not exist in the current scope, the shell tries to retrieve it from a parent scope.
- If you *write* something, that something always is created or modified in the current scope, even if an element of the same name exists in a parent scope.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Observing Scope

- See how scope works in the shell



### Key Points

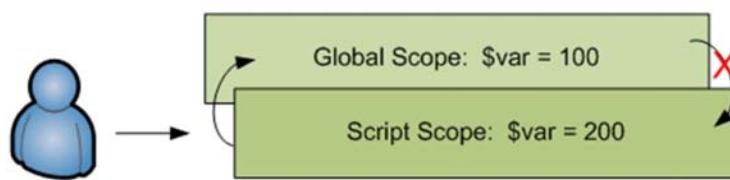
In this demonstration, you will see how scope works in the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Observing Scope.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## General Scope Rules

- Recapping the general rules for scope
  - A child scope can READ elements from its parents
  - A parent scope can never read or write anything in its child scopes
  - You can WRITE only to the current scope
  - If you try to change something in a parent scope, the result is a new item of that same name being created in the current scope



### Key Points

Remember the following general rules:

- A child scope can READ elements from its parents.
- A parent scope can never read or write anything in its child scopes.
- You can WRITE only to the current scope.
- If you try to change something in a parent scope, the result is a new item of that same name being created in the current scope.

## Beyond the Rules

- Cmdlets with scoped elements have a –scope parameter
  - New-Alias      New-PSDrive      New-Variable
  - Set-Variable    Remove-Variable    Clear-Variable
  - Get-Variable
- The scope parameter can take several different values
  - Global means the cmdlet affects the global scope
  - Script means the cmdlet affects a script scope
  - Local means the cmdlet affects the scope in which it is run
  - The number '0' means the current scope
  - The number '1' means the parent scope, and so on

```
New-Variable -name var -value 100 -scope 1
```

### Key Points

Windows PowerShell does not stop you from changing elements in a parent scope. It is, however, generally considered a poor practice. If a script modifies something in the global scope, you cannot tell if that change will have an impact on other things that are running in the shell.

All of the cmdlets that deal with scoped elements have a –scope parameter. These cmdlets include:

- New-Alias
- New-PSDrive
- New-Variable
- Set-Variable
- Remove-Variable
- Clear-Variable
- Get-Variable

The scope parameter can take several different values:

- *Global* means the cmdlet affects the global scope.
- *Script* means the cmdlet affects a script scope. If run within a script, it affects *that* script's scope. If run in a child of a script, it affects the script that the current scope is a child of.
- *Local* means the cmdlet affects the scope in which it is run.
- A number, where 0 is the current scope, 1 is its parent scope, and so on.

If the following command were executed in a script that was run from the command line, the variable would be created in the global scope (one level up from the current scope):

```
New-Variable -name var -value 100 -scope 1
```

MCT USE ONLY. STUDENT USE PROHIBITED

You can also use a special ad-hoc syntax:

- \$global:var refers to the \$var variable in the global scope.
- \$script:var refers to the \$var variable in the current scope (if the current scope is a script), or in the closest parent scope that is a script.
- \$local:var refers to the \$var variable in the current scope.

Again, it is generally considered a poor practice to modify elements outside your current scope because you cannot always be certain of the effects that your modification might have on other tasks or processes.

## Demonstration: Beyond the Rules

- Learn how to use one means of modifying out-of-scope elements



### Key Points

In this demonstration, you will learn how to use one means of modifying out-of-scope elements.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Beyond the Rules.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Best Practices

- Scope can indeed create some very complex and confusing scenarios.
- As a result, some best practices for the use of scope have developed over time.
  - Never refer to a variable until you have explicitly assigned an object to it within the current scope. This practice helps prevent access to parent scope variables, which may contain objects other than those you expected.
  - Never modify an alias, PSDrive, function, or variable in a parent scope. Only modify elements within your current scope.
- These rules apply particularly to scopes outside your own visibility, such as the global scope.
- Try to never modify something that might be different, at a different time, or on a different computer.

## Key Points

To help prevent complex, confusing situations that result in extended debugging, try to observe these best practices:

- Never refer to a variable until you have explicitly assigned an object to it within the current scope. This practice helps prevent access to parent scope variables, which may contain objects other than those you expected.
- Never modify an alias, PSDrive, function, or variable in a parent scope. Only modify elements within your current scope.

These rules apply particularly to scopes outside your own visibility, such as the global scope. Try to never modify something that might be different at a different time, or on a different computer.

## Variable Declaration and Strict Mode

- You do not have to declare variables in advance.
- PowerShell does have the Set-StrictMode cmdlet, which sets a strict mode option.
  - Setting this strict mode option disallows the use of uninitialized variables. Version 2.0 offers functionality that is similar to VBScript's Option Explicit.
  - This option doesn't require advance variable declaration, only that variables must first be assigned a value.
  - Setting strict mode to version 2.0 is a good practice.
- Discuss: Why will this script fail?

```
Set-StrictMode -version 2.0
Function MyFunction {
    Write $var
    $var = 1
    Write $var
}
MyFunction
Write $var
```

### Key Points

You do not have to declare variables in advance. This is a question often asked by administrators who have experience in VBScript because that language provides an Option Explicit command that forces advanced declaration of variables. Windows PowerShell does not have anything that precisely matches Option Explicit; it does, however, have a cmdlet named Set-StrictMode. Set-StrictMode allows you to set a strict mode version, and version 2.0 offers functionality similar to Option Explicit. Here is the command:

```
Set-StrictMode -version 2
```

When in this mode, the shell disallows the use of uninitialized variables. For example, with strict mode off or set to version 1, the following code is permitted:

```
Function MyFunction {
    Write $var
    $var = 1
    Write $var
}
MyFunction
Write $var
```

The following, however, generates errors:

```
Set-StrictMode -version 2.0
Function MyFunction {
    Write $var
    $var = 1
    Write $var
}
MyFunction
Write $var
```

The errors result because the variable \$var is referenced in two places before it has been assigned a value in that scope (you will learn more about scope in the next lesson). The purpose of the error is to help prevent simple syntax errors. For example, consider the following:

```
Function MyFunction {  
    $computer = 'Server2'  
    gwi Win32_ComputerSystem -comp $compter  
}  
MyFunction
```

In this example, the second use of the variable \$computer was misspelled. This script executes but is likely to return an error indicating that WMI could not connect because it is trying to connect to an empty computer name. That error message can be misleading because the correct server name is in the script, and the Get-WmiObject command has been typed accurately apart from the variable name. The following change helps solve the problem:

```
Set-StrictMode -version 2.0  
Function MyFunction {  
    $computer = 'Server2'  
    gwi Win32_ComputerSystem -comp $compter  
}  
MyFunction
```

Now, the shell generates an error about the use of the uninitialized variable \$compter. That error helps to draw your attention more accurately to the actual problem: a mistyped variable name.

Setting strict mode to version 2.0 is a good practice to follow. Doing so helps make certain types of errors more obvious to you, making debugging faster.

Note that a strict mode of 2.0 does not require that you *declare* variables in advance; it prohibits references to variables that *have not yet been assigned a value*. The following example works, without error, with strict mode off:

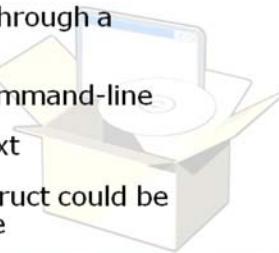
```
Set-StrictMode -off  
Function MyFunction {  
    Write $var  
    $var = 1  
    Write $var  
}  
$var = 3  
MyFunction  
Write $var
```

It works because the first executable line of code assigns the value 3 to \$var. When the function is run, the function is able to draw that value from its parent scope. In other words, \$var has been assigned a value in the script scope and all of its child scopes, and so no error is generated.

## Lesson 3

# Scripting Constructs

- Use the If...ElseIf...Else construct to implement logical decisions
- Use the Switch construct, including parameters such as –wildcard and statements such as Break, to implement logical decisions
- Use the For construct to implement a fixed-iteration loop
- Use the While / Loop / Until constructs to implement variable loops
- Use the ForEach construct to enumerate through a collection of objects
- Use constructs in a script and from the command-line
- Execute two cmdlets on a single line of text
- Identify situations where a scripting construct could be replaced by a pipeline command sequence



Windows PowerShell offers language constructs that are similar to many scripting languages. Its syntax is loosely based on the C# language, although it is similar to other C-based languages such as Java, JavaScript, or PHP.

Many administrators are able to use Windows PowerShell to accomplish complex tasks without ever using these scripting constructs. These constructs are primarily useful when you need a script that implements a complex, multi-step process that requires decision-making or repetitive sub-tasks. These constructs are also helpful for migrating scripts based on older technologies, such as VBScript.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use the If...ElseIf...Else construct to implement logical decisions.
- Use the Switch construct, including parameters such as –wildcard and statements such as Break, to implement logical decisions.
- Use the For construct to implement a fixed-iteration loop.
- Use the While / Loop / Until constructs to implement variable loops.
- Use the ForEach construct to enumerate through a collection of objects.
- Use constructs in a script and from the command line.
- Execute two cmdlets on a single line of text.
- Identify situations where a scripting construct could be replaced by a pipeline command sequence.

## Scripting Constructs

- PowerShell has a relatively small number of scripting constructs. Its constructs enable scripts to take action based on dynamic criteria.
- PowerShell's constructs (that we'll discuss in this lesson):
  - If...ElseIf...Else
  - Switch
  - Break
  - For
  - While / Do / Until
  - ForEach
- **Always remember:** Scripting constructs are not a required feature of the shell. Use only when necessary. Often one-line commands are sufficient.

### Key Points

The half-dozen or so scripting constructs in this lesson constitute Windows PowerShell's formal scripting language. They give your scripts the ability to take different actions based on dynamic criteria, and to execute repetitive tasks. Most of these constructs rely on some comparison criteria, and those comparisons are usually created by using one or more of the shell's comparison operators.

You should remember that Windows PowerShell is intended to meet the needs of a wide audience. Some administrators, for example, do not have a background in programming, and might not be comfortable using these constructs right when they begin learning to use Windows PowerShell. That's fine because the shell still enables them to complete quite powerful and complex tasks *without* using the scripting language portion of the shell. Other administrators, especially those with a programming background or who have been using the shell for a while, may be more comfortable with these scripting constructs. The point is that it's perfectly feasible to use the shell for extremely complex tasks *without* doing anything that you might recognize as programming. These constructs are available for your use if you need them; they are *not* a required feature of the shell. They do, however, enable you to create scripts for ever-more-complex tasks, and so learning them eventually enables you to automate a wider range of administration tasks.

## Reference: If...ElseIf...Else

- If...ElseIf...Else is a decision-making construct
  - It examines one or more logical comparisons, and then executes one or more commands for the first comparison that is true

```
$octet = Read-Host 'Enter the third octet of an IP address'
if ($octet -lt 15 -and $octet -gt 1) {
    $office = 'New York'
} elseif ($octet -ge 15 -and $octet -lt 20) {
    $office = 'London'
} elseif ($octet -ge 21 -and $octet -lt 34) {
    $office = 'Sydney'
} else {
    throw 'Unknown octet specified'
}
```

## Key Points

This is a decision-making construct. It is designed to examine one or more logical comparisons and execute one or more commands for the first comparison that is true.

```
$octet = Read-Host 'Enter the third octet of an IP address'
if ($octet -lt 15 -and $octet -gt 1) {
    $office = 'New York'
} elseif ($octet -ge 15 -and $octet -lt 20) {
    $office = 'London'
} elseif ($octet -ge 21 -and $octet -lt 34) {
    $office = 'Sydney'
} else {
    throw 'Unknown octet specified'
}
```

Some notes regarding this construct:

- You must begin the construct with an if block.
- You may have zero elseif blocks, or as many as needed.
- You may end with an else block.
- Only the first block with a True comparison executes.
- The else block executes only if no other blocks have done so.

## General Construct Notes

- Always remember that a scripting construct's condition must ultimately resolve to either False (0) or True (anything other than 0)
- Although not required, standard formats for readability are commonly used

```
if (5+5) { write 'True' } elseif (5-5) { write 'False' }
```

```
if (5+5)
{
    write 'True'
}
elseif (5-5)
{
    write 'False'
}
```

### Key Points

Scripting constructs' comparison or condition is contained with parentheses. The comparison or condition can be as complex as needed and can contain parenthetical subexpressions. However, the entire condition must ultimately resolve to either True or False.

Note that Windows PowerShell internally represents False as zero and True as any nonzero number. So if your condition results in zero, it is considered False. The following construct, then, would display True:

```
if (5+5) {
    write 'True'
} elseif (5-5) {
    write 'False'
}
```

The conditional code—that is, the code or commands that executes when the comparison is True—is contained within curly braces. The shell isn't picky about the formatting of these braces. The following, for example, is completely permissible:

```
if (5+5) { write 'True' } elseif (5-5) { write 'False' }
```

However, that example isn't very easy to read. As a best practice, you should indent the conditional code and adopt a standard for where braces are placed. One common standard is to place the first brace on the line following the condition and the closing brace on a line after the last conditional statement or command. That is the format that has been used for most examples in this course thus far and the format that will be used going forward. Other commonly used formats include the following:

```
if (5+5)
{
    write 'True'
}
elseif (5-5)
```

```
{  
    write 'False'  
}
```

It doesn't matter what format you choose, as long as you indent the conditional code and remain consistent with your placement of the curly braces.

MCT USE ONLY. STUDENT USE PROHIBITED

## Nested Constructs

- You often need to nest constructs within each other
  - The only rule is that they must be entirely nested
  - The inner construct must be close before the outer construct
- Discuss: Where in the code below are the inner and outer If constructs?

```
If ($ping.pingstatus -eq 0) {  
    $wmi = gwmi win32_operatingsystem -computer $computer  
    if ($wmi.buildnumber -ne 7600) {  
        write 'Incorrect build - ignoring computer'  
    }  
}
```

### Key Points

All constructs may be nested within one another. The only rule is that they be *entirely* nested. That is, the inner construct must be closed before the outer construct. Indenting each conditional level helps you visually check to make sure you have done so; many third-party script editors provide additional visual cues to help you ensure that you are properly nesting your constructs.

Here is an example of a nested If construct:

```
If ($ping.pingstatus -eq 0) {  
    $wmi = gwmi win32_operatingsystem -computer $computer  
    if ($wmi.buildnumber -ne 7600) {  
        write 'Incorrect build - ignoring computer'  
    }  
}
```

Notice how the indentations help to set the nested construct aside separately. Windows PowerShell does not enforce this, although the Windows PowerShell ISE does facilitate this kind of formatting. The following is completely permissible, although it is harder to read and maintain:

```
If ($ping.pingstatus -eq 0) {  
    $wmi = gwmi win32_operatingsystem -computer $computer  
    if ($wmi.buildnumber -ne 7600) {write 'Incorrect build - ignoring computer'}}}
```

## Typing Constructs

- Scripting constructs needn't necessarily be parts of a true script
  - With PowerShell, you can run the same commands interactively in the shell's command line
  - This is useful for small-scale applications of scripting constructs that you create directly at the command line
- An example of typing in a script at the command line:

```
PS C:\> if ($var -eq 5) {  
    >> write 'is 5'  
    >> } else {  
    >> write 'is not 5'  
    >> }  
>>  
is not 5  
PS C:\>
```

### Key Points

When Windows PowerShell runs a script, it does so exactly as if you had manually typed each line of that script into the interactive shell. That is, there are no differences between running a script and running the same commands interactively, save that a script runs only if the shell's execution policy permits it. Even with the execution policy set to Restricted, you could easily open a script, copy its contents to the Clipboard, and then paste those contents into the interactive shell.

**Note:** Being able to copy and paste a script into the shell isn't considered a security problem because you can't do it without taking several deliberate actions. There's really no practical way to accidentally copy and paste commands as described.

That said, you can use any of the scripting constructs directly from the command line, even in the console window rather than the ISE. When you open a construct by typing { and pressing Return, the shell's prompt changes, indicating that you are inside a construct and that the shell is waiting for you to complete it:

```
PS C:\> if ($var -eq 5) {  
    >>
```

When you finish typing the construct and close it, press Return one more time to execute the code:

```
PS C:\> if ($var -eq 5) {  
    >> write 'is 5'  
    >> } else {  
    >> write 'is not 5'  
    >> }  
>>  
is not 5
```

```
PS C:\>
```

When including this kind of construct in a cmdlet script block, you may wish to forgo best practices-style formatting in favor of something more suited to a single-line command. For example:

```
gwmi win32_logicaldisk | select deviceid, @{Label='DriveType';Expression={ if ($_.drivetype -eq 3) { write 'LocalFixed' } else { write 'NonFixed' }}}}, Size
```

This one-line command includes a `Select-Object` cmdlet that uses a custom column. The expression for that custom column includes an If construct, but it is not formatted according to best practices. The output of this command looks something like the following:

deviceid	DriveType	Size
A:	NonFixed	
C:	LocalFixed	64317550592
D:	NonFixed	
NonFixed	413256384512	Z:

As you can see, the scripting construct could be used entirely from the command line, and not in a script at all.

## Demonstration: Working with Constructs

- Learn how to use constructs directly at the command line



### Key Points

In this demonstration, you will learn how to use constructs directly at the command line.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Working with Constructs.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Reference: Switch

- Switch is also designed to make logical decisions. It compares a single input object to a range of possible objects or comparisons.

```
$computer = 'LON-DC1'
switch ($computer) {
    'LON-DC1' {
        write 'London Domain Controller 1'
    }
    'SEA-DC1' {
        write 'Seattle Domain Controller 1'
    }
    'LON-DC1' {
        write 'London Domain Controller 1'
    }
    default {
        write 'Unknown'
    }
}
```

### Key Points

This construct is also designed to make logical decisions. It compares a single input object to a range of possible objects or comparisons. By default, it executes each matching condition, not just the first one. Here is an example:

```
$computer = 'LON-DC1'
switch ($computer) {
    'LON-DC1' {
        write 'London Domain Controller 1'
    }
    'SEA-DC1' {
        write 'Seattle Domain Controller 1'
    }
    'LON-DC1' {
        write 'London Domain Controller 1'
    }
    default {
        write 'Unknown'
    }
}
```

Some notes for this construct:

- Note in the example that the output includes *London Domain Controller 1* twice because there are two matching conditions.
- The default block is optional and executes only if no other blocks have matched. It must be the last block if it is used.
- Each possible match is its own subconstruct and must enclose its code in curly braces.

## Reference: The Break Keyword

- Break automatically exits any construct except If...ElseIf...Else. Use Break with Switch to ensure that only the first matching condition is executed.

```
$computer = 'LON-DC1'
switch ($computer) {
    'LON-DC1' {
        write 'London Domain Controller 1'
        break
    }
    'SEA-DC1' {
        write 'Seattle Domain Controller 1'
        break
    }
    'LON-DC1' {
        write 'London Domain Controller 1'
        break
    }
    default {
        write 'Unknown'
    }
}
```

## Key Points

The Break keyword immediately exits any construct *except* an If...ElseIf...Else construct. You can use Break in conjunction with Switch to ensure that only the first matching condition is executed.

```
$computer = 'LON-DC1'
switch ($computer) {
    'LON-DC1' {
        write 'London Domain Controller 1'
        break
    }
    'SEA-DC1' {
        write 'Seattle Domain Controller 1'
        break
    }
    'LON-DC1' {
        write 'London Domain Controller 1'
        break
    }
    default {
        write 'Unknown'
    }
}
```

The output of this example is a single *London Domain Controller 1* because the Break keyword immediately exits the construct as part of the first matching condition.

## Reference: Switch Options

- Switch includes several options to modify its behavior, such as –wildcard, which enables wildcard string matches

```
$computer = 'LON-DC1'
switch -wildcard ($computer) {
    'LON*' {
        write 'London'
    }
    'SEA*' {
        write 'Seattle Domain Controller 1'
    }
    '*DC*' {
        write 'Domain Controller'
    }
    '*SRV*' {
        write 'Member Server'
    }
    default {
        write 'Unknown'
    }
}
```

### Key Points

The Switch construct includes several options that modify its behavior. One of these is the –wildcard switch, which enables wildcard string matches. In the following example, the output would be "London" and "Domain Controller." Notice that the break keyword is not used so that multiple conditions may execute.

```
$computer = 'LON-DC1'
switch -wildcard ($computer) {
    'LON*' {
        write 'London'
    }
    'SEA*' {
        write 'Seattle Domain Controller 1'
    }
    '*DC*' {
        write 'Domain Controller'
    }
    '*SRV*' {
        write 'Member Server'
    }
    default {
        write 'Unknown'
    }
}
```

To see other capabilities and options for Switch, run Help about\_switch.

## Reference: For

- For is used to repeat one or more commands a given number of times

```
For ($i=0; $i -lt 10; $i++) {  
    Write $i  
}
```

### Key Points

This construct is normally used to repeat one or more commands a given number of times.

```
For ($i=0; $i -lt 10; $i++) {  
    Write $i  
}
```

Some notes for this construct:

- The construct's comparison or condition is actually three parts, separated by semicolons:
  - The first part sets a starting condition, in this case setting \$i=0.
  - The second part is the condition that keeps the loop going, in this case so long as \$i is less than 10.
  - The third part is an operation that is executed each time the loop completes, in this case incrementing \$i by one.

It is permissible to modify the counter variable within the loop itself:

```
For ($i=0; $i -lt 10; $i++) {  
    Write $i  
    $i = 10  
}
```

This example would execute the loop only once because during that execution, \$i would be set outside the range (less than 10) that keeps the loop repeating.

Keep the shell's scope rules in mind. The following is a poor practice:

```
For ($i; $i -lt 10; $i++) {  
    Write $i
```

```
}
```

In this example, \$i is identified as the loop's counter variable, but it is not assigned a starting value. The shell is forced to go up the scope hierarchy to see if \$i was defined in any parent scope. If it was not, \$i is given a default value of zero. However, if \$i was created in any parent scope, \$i may have a nonzero value or may even contain an object other than an integer. This can result in unexpected behavior and can be tricky to debug if you aren't paying attention. As a best practice, *always* assign a starting value to the counter variable.

MCT USE ONLY. STUDENT USE PROHIBITED

## Reference: While / Do / Until

- While / Do / Until are used in different combinations to create slightly different effects

```
$var = 1
while ($var -lt 10) {
    write $var
    $var++
}

$var = 1
do {
    write $var
    $var++
} while ($var -lt 10)

$var = 20
do {
    write $var
    $var--
} until ($var -lt 10)
```

### Key Points

These keywords can be used in different combinations to create slightly different effects. Here are three variations:

```
$var = 1
while ($var -lt 10) {
    write $var
    $var++
}

$var = 1
do {
    write $var
    $var++
} while ($var -lt 10)

$var = 20
do {
    write $var
    $var--
} until ($var -lt 10)
```

Some notes for these constructs:

- When the comparison or condition occurs at the end of the construct, the conditional code within the construct *always* executes at least one time.
- When the comparison or condition occurs at the beginning of the construct, the construct's conditional code executes only if the comparison is True to begin with. In some situations, that may mean that the conditional code does not execute at all.
- Although While may appear at either the beginning or the end of the construct, Until may be used only at the end.
- While and Until do essentially the same thing at the end of a construct but with opposite logic.

## Reference: ForEach

- ForEach performs the same basic task as the ForEach-Object cmdlet. It enumerates through a collection of objects, executing its configured code once for each.
  - ForEach uses the 'in' keyword rather than the \$\_ placeholder.

```
$services = Get-Service
ForEach ($service in $services) {
    Write $service.name
}
```

- Remember: ForEach and ForEach-Object are often completely unnecessary!

### Key Points

This construct performs the same basic task as the ForEach-Object cmdlet: It enumerates through a collection of objects. The construct's conditional code executes one time for each object in the input collection. During each execution of the loop, the next object is taken from the collection and placed into a variable that you designate. For example:

```
$services = Get-Service
ForEach ($service in $services) {
    Write $service.name
}
```

Notice that the syntax is slightly different from the ForEach-Object cmdlet. This construct does not use the \$\_ placeholder, and its condition uses the in keyword.

This difference can be especially confusing because the shell defines an alias, ForEach, for the ForEach-Object cmdlet. That means "foreach" is both a scripting construct and an alias/cmdlet! The shell can tell the difference based on where they are used. The alias would be used only in a command-line sequence and would typically have objects being piped into it.

```
Get-Service | ForEach { Write $_.name }
```

It's important not to mix up these two different forms of syntax.

### Do You Really Need ForEach?

In many cases, administrators with a background in programming—such as using VBScript—use ForEach-Object or a ForEach construct when it is not strictly necessary. There's nothing wrong with doing so, but it is a bit of extra work. For example, consider this code snippet, which deletes all of the files in a given folder if they are over a certain size:

```
$folder = 'c:\demo'
```

```
$files = dir $folder
foreach ($file in $files) {
    if ($file.length -gt 100MB) {
        del $file
    }
}
```

This is a very script-style approach and one that you would see used frequently in languages like VBScript. Windows PowerShell, however, enables a much easier model because most cmdlets can handle collections of objects without requiring you to enumerate through them:

```
Dir c:\demo | where { $_.Length -gt 100MB } | Del
```

Any time you find yourself using `ForEach` or `ForEach-Object`, you may want to take a moment to consider whether or not it is necessary. In some cases, it absolutely is, as you learned in the module on Windows Management Instrumentation. However, in many other instances, it is not necessary to enumerate objects yourself.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Writing Constructs

- Learn how to write a variety of constructs



### Key Points

In this demonstration, you will learn how to write a variety of constructs.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Writing Constructs.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## On the Command Line

- It may be easier to eliminate script formatting when typing at the command line
  - Although PowerShell can and will recognize formatting when used during command line entry
  - Both of these work:

```
gwmi win32_logicaldisk | select deviceid,
@{Label='DriveType';Expression={ switch
($_.drivetype) { 2 { Write 'Floppy'; break } 3 { Write
'Fixed'; break } 4 { Write 'Optical'; break } 5 { Write
'Network'; break }}}}, Size, FreeSpace
```

```
PS C:\> gwmi win32_logicaldisk |
>> select deviceid, @{Label='DriveType';Expression={
>>   switch ($_.drivetype) {
>>     2 { Write 'Floppy'; break }
>>     3 { Write 'Fixed'; break }
...
...}}
```

### Key Points

When typing constructs on the command line, perhaps as part of a script block being passed to a cmdlet, you may want to forgo the formal formatting of a script in favor of brevity.

In those cases, you may need to type multiple commands on a single physical line. Windows PowerShell permits you to do so, provided the individual command sequences are separated by semicolons:

```
gwmi win32_logicaldisk | select deviceid, @{Label='DriveType';Expression={ switch
($_.drivetype) { 2 { Write 'Floppy'; break } 3 { Write 'Fixed'; break } 4 { Write
'Optical'; break } 5 { Write 'Network'; break }}}}, Size, FreeSpace
```

This command outputs a human-readable drive type, based on the numeric drive type contained within the WMI class. Notice the semicolon used to separate the Write and Break commands within each conditional block. Although this can be typed and run exactly as shown, it is easier to read when some formatting is applied:

```
gwmi win32_logicaldisk |
  select deviceid, @{Label='DriveType';Expression={
    switch ($_.drivetype) {
      2 { Write 'Floppy'; break }
      3 { Write 'Fixed'; break }
      4 { Write 'Optical'; break }
      5 { Write 'Network'; break }
    }
  }
}, Size, FreeSpace
```

The second example could also be typed exactly as-is because the shell knows how to parse that kind of input. Typing it into the shell would look something like this:

```
PS C:\> gwmi win32_logicaldisk |
>> select deviceid, @{Label='DriveType';Expression={
```

```
>>     switch ($_.drivetype) {
>>         2 { Write 'Floppy'; break }
>>         3 { Write 'Fixed'; break }
>>         4 { Write 'Optical'; break }
>>         5 { Write 'Network'; break }
>>     }
>> },
>> ], Size, FreeSpace
>>
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Changing Your Thinking

- Remember: PowerShell doesn't need to be a scripting language.
  - If you've worked with script languages before, you may tend to naturally approach PowerShell in that way.
  - There is nothing wrong with this approach.
  - However, PowerShell is primarily an administrative automation solution. There are usually command sequences that accomplish tasks and don't require true scripting.
- Taking the time to "change your thinking" and adopt the PowerShell-centric way of accomplishing tasks
  - Makes you more efficient.
  - Accomplishes your needed tasks more quickly.
  - Enhances your ability to construct command sequences on the fly, fix problems, and get the job of IT done much better!

### Key Points

If you've worked with a scripting language before, you may tend to naturally approach Windows PowerShell using the same techniques that you are accustomed to using in another language. There's nothing wrong with that, and the presence of scripting constructs within the shell is in part designed to help facilitate the transfer of your old skills into this new environment.

However, keep in mind that Windows PowerShell often enables you to accomplish the same thing with much less work. Taking the time to change your thinking and adopt a more Windows PowerShell-centric way of accomplishing tasks, can make you not only more efficient, but in some cases it can enable new capabilities.

As a general rule, try to accomplish tasks by using command sequences rather than formal scripts. For very complex tasks, a script may of course be the right way to get the job done, but in many cases, a simpler command-line sequence can do the job.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Commands, Not Scripts

- Learn how to write a command that is equivalent to an entire script



### Key Points

In this demonstration, you will learn how to write a command that is equivalent to an entire script.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Commands Not Scripts.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lab B: Using Scripting Constructs

- Exercise 1: Processing and Validating Input
- Exercise 2: Working with For, While, Foreach, and Switch
- Exercise 3: Using the Power of the One-Liner

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2	LON-CLI1
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

Some of your junior IT staff could benefit from PowerShell scripts but they don't have the skills yet to write them themselves. You want to create some command-line utilities that allow those staff members to do specific tasks based on their input. As an Active Directory administrator, you need to be able to provision new accounts and make large scale changes very easily. PowerShell looping constructs have become required knowledge for you to get this work done in an efficient manner. You want to be careful with your scripts so that they don't take too long to run in your enterprise and so that they don't use too many system resources.

Before you begin the lab you must:

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines. Wait until the boot process is complete.
2. Log on to LON-DC1 with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open the Windows PowerShell ISE on LON-DC1. All PowerShell commands are run within the Windows PowerShell ISE program.
4. Disable the firewall on the LON-DC1, LON-SVR1, LON-SVR2, and LON-CLI1 virtual machines.

This is not a security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on PowerShell concepts.

MCT USE ONLY. STUDENT USE PROHIBITED

**Note:** You may choose to complete these exercises in the Windows PowerShell ISE or in the Windows PowerShell console. In the console, you can open a construct using a left brace {, and the shell allows you to press Enter to continue a new line. When you do so, the shell prompt changes to >>, indicating that you are still inside a construct. Continue typing, and when you are done, close the construct with a right brace } and press Enter. You must press Enter one last time on a >> line to tell the shell that you are finished.

Alternately, complete the exercises in the Windows PowerShell ISE, which is better-designed for multiline commands and scripts.

## Exercise 1: Processing and Validating Input

### Scenario

Some of your junior IT staff could benefit from PowerShell scripts but they don't have the skills yet to write them themselves. You want to create some command-line utilities that allow those staff members to do specific tasks based on their input.

The main tasks for this exercise are as follows:

1. Read input from the end user.
2. Validate input received from the end user.
3. Create a script that prompts for specific input and exits only when the right input is found or when no input is provided.

#### ► Task 1: Read input from the end user

1. Read the help documentation and examples for the **Read-Host** cmdlet so that you understand how it is used.
2. Use **Read-Host** to prompt for input, setting the prompt text to *Enter anything you like and press <ENTER> when done.*
3. Create a new variable called **serverName** and assign a value entered at the prompt through **Read-Host**. Use *Server Name* for the prompt text.

#### ► Task 2: Validate input received from the end user

1. Read the documentation in **about\_if** to understand how **if** statements work.
2. Create an **if** statement that outputs *Server name '\$serverName' is REJECTED* if the value of the **serverName** variable does not start with *SERVER* or *Server name '\$serverName' is OK!* if the value of the **serverName** variable does start with *SERVER*.

#### ► Task 3: Create a script that prompts for specific input and exits only when the right input is found or when no input is provided

1. Read the documentation in **about\_do** to understand how the **do...while** statement works.
2. Create a **do...while** statement that clears the screen and then prompts the user for a server name. The prompt text should be '*Server Name (this must start with "SERVER")*'. The loop should terminate when the user either provides a server name that starts with *SERVER* or doesn't provide any input at all and just presses *<ENTER>*.
3. Output the value of the **serverName** variable in the console.
4. Combine the results of steps 2 and 3 into a script and run it.

**Results:** After this exercise, you should be able to read and process user input in a PowerShell script.

## Exercise 2: Working with For, While, Foreach, and Switch

### Scenario

As an Active Directory administrator, you need to be able to provision new accounts and make large-scale changes very easily. PowerShell looping constructs have become required knowledge for you to get this work done in an efficient manner. You'll start with some basic for a while loops, and then move on to using what you've learned for your Active Directory tasks.

The main tasks for this exercise are as follows:

1. Create a basic for loop.
2. Create a basic while loop.
3. Use the for statement to create users in Active Directory.
4. Use foreach to iterate through a collection of objects.
5. Use switch to test a value and translate it to a human-readable string.

#### ► Task 1: Create a basic for loop

1. Create a new variable called **i** and assign a value of **100** to that variable.
2. Read the documentation in **about\_For** so that you understand how **for** loops work.
3. Create a **for** loop with the following values:

init: **\$i**  
 condition: **\$i -lt 100**  
 repeat: **\$i++**

Inside the **for** loop write the value of **\$i** out to the current host using **Write-Host**. Note that the loop does not run and no output is produced.

4. Repeat the last task using an init value of **\$i = 0**. Note that the numbers 0 through 99 are output to the host.

#### ► Task 2: Create a basic while loop

1. Assign a value of 0 to the **i** variable.
2. Read the documentation in **about\_While** so that you understand how **while** loops work.
3. Create a **while** loop with the following condition: **\$i -lt 100**. Inside the **while** loop write the value of **\$i** out to the current host using **Write-Host** and increment the value of **\$i**. Note that this loop produces the exact same output as the **for** loop you just ran.

#### ► Task 3: Use the for statement to create users in Active Directory

1. Import the Active Directory module into the current session.
2. Read the help documentation and examples for the **New-ADUser** cmdlet so that you understand how it works.
3. Create a new user named **Module11BTest1** in the **CN=Users,DC=contoso,DC=com** container with a SAM account name of **Module11BTest1**. Use the **PassThru** parameter to show the user once they are created.
4. Create a new variable called **numUsers** and assign the value of **10** to that variable.
5. Create a **for** loop to generate 9 more users. The **for** loop counter (**\$i**) should be initialized to **2** and the **for** loop should terminate when the counter is equal to the value of the **numUsers** variable. All users should have a name and SAM account name of **Module11BTest\$i** and they should be created

- in the **CN=Users,DC=contoso,DC=com** container. Use the **PassThru** parameter to show the users once they are created.
6. Remove the test users by retrieving all users whose name starts with **Module11BTest** using **Get-ADUser** and piping them to **Remove-ADUser**.

► **Task 4: Use foreach to iterate through a collection of objects**

1. Read the documentation in **about\_foreach** so that you understand how the **foreach** statement works.
2. Retrieve a list of logical disks using **Get-WmiObject** with the **Win32\_LogicalDisk** class. Store the results in a new variable called **disks**.
3. Use the **foreach** operator to iterate through the **disks** array. The current item should be stored in a variable called **disk**. Inside the loop use **Select-Object** to output the **DeviceID** and **DriveType** properties for each disk.

► **Task 5: Use switch to test a value and translate it to a human-readable string**

1. Read the documentation in **about\_switch** so that you understand how the **switch** statement works.
2. Using the same **foreach** loop you just created, add a **switch** statement to lookup a display name for the **DriveType** property integer value. Store the display name in a variable called **driveTypeValue**. Update the **Select-Object** statement to output the **driveTypeValue** value as a calculated value called **DriveType** instead of the **DriveType** property. The **DriveType** values and their corresponding display names are:

- 0: Unknown
- 1: No Root Directory
- 2: Removable Disk
- 3: Local Disk
- 4: Network Drive
- 5: Compact Disk
- 6: RAM Disk

**Results:** After this exercise, you should be able to use the for, foreach, and switch statements in your PowerShell scripts.

## Exercise 3: Using the Power of the One-Liner

### Scenario

Now that you can perform large scale optimizations against Active Directory, you want to be careful with your scripts so that they don't take too long to run in your enterprise and so that they don't use too many system resources. At the same time, you also want to prove to yourself how scripting and one-liner commands can be interchangeable.

The main tasks for this exercise are as follows:

1. Create scripts to retrieve and process data.
2. Create one-liners for those scripts that are much more efficient.

#### ► Task 1: Create scripts to retrieve and process data

1. Use the **foreach** statement to iterate through a list of the names of all computers in your domain (**LON-DC1**, **LON-SVR1**, **LON-SVR2** and **LON-CLI1**). For each computer, get the **Windows Update** service using **Get-Service**.
2. Use the **foreach** statement to iterate through all Active Directory users. For each user, include the **Office** attribute in the results and show only users that have an Office attribute value of *Bellevue*.

#### ► Task 2: Create one-liners for those scripts that are much more efficient

1. Use **Get-Service** to retrieve the Windows Update service from all computers in the domain. Pass an array of computer names directly into the **ComputerNames** parameter.
2. Use **Get-ADUser** to retrieve all Active Directory users whose **Office** attribute has a value of *Bellevue*.

**Results:** After this exercise, you should understand that many short scripts can be converted into simple one-liner commands that are efficient and require less memory.

## Lab Review

1. Name 3 different keywords that allow you to do something in a loop.
2. What keyword should you use when you need to compare one value against many other individual values?
3. What makes a PowerShell one-liner so powerful?
4. What is the name of the top-most scope?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 4

# Error Trapping and Handling

- Explain the purpose of the \$ErrorActionPreference variable and the use of the –ErrorAction and –ErrorVariable parameters
- Write a trap construct to handle an error
- Write a Try...Catch construct to handle an error



Although you should catch and fix certain errors—such as typos—as you create your scripts, other types of errors can be anticipated only when you are writing a script. For example, the error that results from a file being unavailable, or from a server being unavailable, are errors that you can *anticipate* in advance but cannot actually fix or prevent. What you can do, however, is write scripts that can detect those errors and handle them gracefully.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of the \$ErrorActionPreference variable and the use of the –ErrorAction and –ErrorVariable parameters.
- Write a trap construct to handle an error.
- Write a Try...Catch construct to handle an error.

MCT USE ONLY. STUDENT USE PROHIBITED

## Types of Errors in PowerShell

- **Syntax errors**, which are usually the result of a mistype or misspelling
  - Easiest to prevent and fix, usually identified through runtime error messages
- **Logic errors**, which means the script executes without error, but does not accomplish the results you intended
  - Difficult to fix, and can require debugging
  - Often caused by variable or property values that contain unexpected information
- **Run-time errors**, which are errors that you can anticipate in advance but cannot necessarily fix in advance
  - Less difficult to fix, and can often be due to conditions in the environment at runtime, such as a computer that isn't available on the network

### Key Points

There are three broad categories of errors in a Windows PowerShell script:

- **Syntax errors**, which are usually the result of a mistype or misspelling
- **Logic errors**, which means your script executes without error, but does not accomplish the results you intended
- **Run-time errors**, which are errors that you can anticipate in advance—such as a computer that isn't available on the network—but cannot necessarily fix in advance

Syntax errors are, relatively speaking, the easiest to prevent and to fix. Error messages usually take you to the right location within your script to locate and correct the error, and you just need to have a strong attention to detail to catch minor typographical problems.

Logic errors are more difficult, and require debugging—which is something you will learn in the next lesson. Logic errors commonly result from a variable or property value that contains information other than what you expected or assumed, such as a *DriveType* property that you thought contained a string such as "Removable," but which in fact contains a number like "5."

Finally, run-time errors fall somewhere in between syntax and logic errors in terms of how difficult they are to deal with. Run-time errors cannot always be prevented in advance. For example, you might write your script to text connectivity to a remote computer before trying to connect to it, but doing so does not prevent an error if your script tries to connect and doesn't have the necessary permissions. This lesson focuses on these types of run-time errors, which you cannot always prevent but which you can *anticipate* as you write your scripts.

## Discussion: Run-time Errors

- What kinds of errors could you foresee being able to anticipate within a script?
  - Connectivity to computers
  - Permissions to perform some task
  - Files, users, or other objects not being found
  - What else?



### Key Points

What kinds of errors could you foresee being able to anticipate within a script?

- Connectivity to computers
- Permissions to perform some task
- Files, users, or other objects not being found
- What else?

## Error Actions

- Many cmdlets can continue to execute, even after certain types of nonterminating errors occur
- The \$ErrorActionPreference shell variable instructs the shell what to do when a nonterminating error occurs
- Four possible values for \$ErrorActionPreference
  - **Continue** is the default, and instructs the cmdlet to display an error message and continue executing
  - **Stop** tells the cmdlet to turn the nonterminating error into a *terminating exception*, meaning the cmdlet stops execution entirely
  - **SilentlyContinue** instructs the cmdlet to continue executing without displaying an error message
  - **Inquire** tells the cmdlet to interactively prompt the user for an action, enabling the user to select either Continue or Stop

### Key Points

Many cmdlets are capable of continuing to execute when certain types of errors occur. For example, consider this command:

```
Get-WmiObject Win32_Service -computer Server1,Server2,Server3
```

Suppose that Server2 is temporarily offline when this command is run. The cmdlet is successful in contacting Server1, but Server2 fails. This is a *nonterminating error*. The cmdlet can't perform its action for Server2, but that doesn't stop it from moving on to try the next server, Server3.

When a cmdlet encounters a non-terminating error, it checks to see what it should do about it. The non-terminating error action is contained in a built-in shell variable called \$ErrorActionPreference. There are four possible values for this variable:

- **Continue** is the default, and instructs the cmdlet to display an error message and continue executing.
- **Stop** tells the cmdlet to turn the non-terminating error into a *terminating exception*, meaning the cmdlet stops execution entirely.
- **SilentlyContinue** instructs the cmdlet to continue executing without displaying an error message.
- **Inquire** tells the cmdlet to interactively prompt the user for an action, enabling the user to select either Continue or Stop.

Remember that these apply only to the nonterminating errors than a cmdlet might encounter. Some cmdlets can encounter terminating exceptions, and when that happens they always stop running immediately, regardless of how \$ErrorActionPreference is set.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Errors in Action

- Learn how to use \$ErrorActionPreference



### Key Points

In this demonstration, you will learn how to use \$ErrorActionPreference.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Errors in Action.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Poor Practice

- Remember: Errors are a good thing
  - Without errors, you will never discover where and when your script is inappropriately executing
- Some administrators have begun to add this command to the beginning of their scripts to prevent errors from occurring:

```
$ErrorActionPreference = "SilentlyContinue"
```

- This is not a good practice
  - Some administrators believe that this setting allows the script to continue when run-time errors occur, which it does
  - However, a much better practice is to create logic that surrounds and anticipates expected run-time errors, rather than eliminating errors altogether

## Key Points

In general, error messages are a good thing. The shell typically produces informative, clear error messages that—if you read them carefully—tell you what's going wrong and give you clues toward finding a fix for the problem. You do not, generally speaking, want to globally suppress all error messages.

However, it is becoming a common practice for administrators to add this command to the beginning of a script:

```
$ErrorActionPreference = "SilentlyContinue"
```

Administrators who do this typically do so because they have a cmdlet—such as Get-WmiObject—that they anticipate producing an error, and they want to suppress the error message and allow the cmdlet to continue executing. But adding the preceding setting to a script suppresses *all* errors for the *entire* script. In many cases, the script runs into other problems that the administrator would want to know about, but because they have suppressed all error messages, no errors are displayed. The result is a script that does not run as intended, and the administrator spends a great deal of time attempting to debug the script because the shell is unable to provide useful clues about what the problem is.

There are very few instances where you would want to suppress all error messages from an entire script. Instead, you commonly want to suppress the errors from a single cmdlet, and you can do so.

## More Granular Error Actions

- All PowerShell cmdlets support a set of common parameters
  - These parameters are handled directly by the shell, and do not need to be specifically written by the cmdlet's developer
  - These parameters are not listed in each cmdlet's help file
  - The cmdlet's help file may simply include *common parameters* at the end of its parameter list
- Common parameters can be discovered by using:  
**help about\_commonparameters**
- One common parameters is –ErrorAction, which accepts the same four values as \$ErrorActionPreference
  - Used for controlling error actions for a single cmdlet, rather than globally for the shell session

### Key Points

All Windows PowerShell cmdlets support a set of *common parameters*, which are actually handled directly by the shell and do not need to be specifically written by the cmdlet's developer. These parameters are not listed in each cmdlet's help file; rather, you'll see the cmdlet's help file include Common Parameters at the end of the cmdlet's parameter list.



Review the help file for any cmdlet. Do you see the Common Parameters reference at the end of the parameter list?

You can read about these common parameters by running:

```
Help about_commonparameters
```

One of the common parameters is –ErrorAction, which has the global parameter alias –EA. This parameter accepts the same four possible values as \$ErrorActionPreference. However, rather than globally controlling the error action, the parameter controls the error action for a single cmdlet. So if you have a cmdlet—such as Get-WmiObject—and you anticipate it causing errors, you can suppress the non-terminating errors for *just that cmdlet* by specifying –ErrorAction or –EA:

```
Gwmi Win32_Service -computer Server1,Server2,Server3  
-EA SilentlyContinue
```

## Demonstration: -ErrorAction

- Learn how to use the `-ErrorAction` parameter



### Key Points

In this demonstration, you will learn how to use the `-ErrorAction` parameter.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\ErrorAction.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Trapping Exceptions

- You can instruct the shell to notify you of terminating exceptions, enabling you to issue commands in response to those exceptions
- This is called *error trapping*
  - You can trap only a terminating exception
  - You cannot trap a nonterminating error, even if an error message is displayed
- Enabling trapping begins by setting the Stop value for –ErrorAction
- Once enabled with –ErrorAction Stop, two choices are available for actually trapping the error:
  - A Trap construct
  - A Try...Catch construct

### Key Points

It is possible to have the shell notify you of terminating exceptions, and enable you to issue commands in response to those exceptions. This is called *error trapping*, meaning that you define your own action for the exception, rather than allowing the shell to perform its default action.

**Note:** You can trap only a terminating exception. You cannot trap a nonterminating error, even if an error message is displayed.

If you anticipate a cmdlet running into an error condition that you want to trap, you *must* specify the –ErrorAction parameter and use the Stop value. Any other error action will not guarantee a trappable exception.

**Note:** If you use –ErrorAction Inquire, a trappable exception is generated only if an error occurs and the user selects the option to stop the cmdlet.

Once you have instructed a cmdlet to turn non-terminating errors into terminating, trappable exceptions, by means of –EA Stop, you have two choices for actually trapping the error: A Trap construct, or a Try...Catch construct.

## Exception Scope

- Recall: A PowerShell session and its scripts work within multiple scopes

- The shell itself operates in the global scope
  - Each script you write typically creates its own scope
  - Running scripts and function creates a scope tree
  - Rules exist for reading and writing between scopes



- Exceptions respect scope rules
  - You should trap exceptions in the same scope they occur

## Key Points

Before you move on to Trap and Try...Catch constructs, take a moment to review what you know about scope within Windows PowerShell.

The shell itself is referred to as the *global* scope. Each script that you run normally creates its own *script* scope. Functions (which we will cover later in this module) are also contained in their own scope. So if you execute a script, which in turn contains a function, then you have a scope tree that looks something like this:

- Global Scope
  - Script scope
    - Function scope

The function's scope is a child of the script, which is a child of the global. If a terminating exception occurs within the function, the shell first checks to see whether you are attempting to trap and handle that exception *within the same scope*—that is, within the function itself. If you have not provided a means to do so, the shell exits the function's scope, passing the terminating exception to the parent scope—in this case, the script. From the script's point of view, the function itself generated the exception, and the shell now checks to see whether the script contains something to trap and handle the exception. If it does not, the shell exits the script's scope, passing the exception to its parent global scope. From the global scope's perspective, the entire script generated the exception, and the shell checks to see whether any error trapping has been implemented in the global scope. You don't usually do so in the global scope, and because there is no parent scope to pass the exception to, the shell takes its default behavior and displays the exception as an error message.

This can seem a bit complicated, but it is very important to decide how you will deal with exceptions. Generally speaking, you should trap and handle exceptions in the same scope in which they occur. If a function contains Get-WmiObject, for example, and you want to trap and handle errors for that cmdlet,

your error trapping construct should go into the function. That way, the shell stays within the function's scope as you handle the exception.

MCT USE ONLY. STUDENT USE PROHIBITED

## Trap Constructs

- Trap constructs are usually defined in scripts before the anticipated exception. A simple trap construct might resemble:

```
trap {  
    write-host "Exception trapped!" -fore yellow -back black  
    continue  
}  
get-wmiobject win32_process -comp NotOnline -ea stop
```

- At the end of a Trap construct, you will specify one of two available keywords:
  - Continue** will resume execution on the command *following* the one that caused the exception, staying within the same scope as the trap
  - Break** will exit the scope that contains the trap, passing the original exception to the parent scope for handling

### Key Points

A trap construct is usually defined in a script *before* the exception that you are anticipating. A simple trap construct might look like this:

```
trap {  
    write-host "Exception trapped!" -fore yellow -back black  
    continue  
}  
get-wmiobject win32_process -comp NotOnline -ea stop
```

At the end of the trap construct, you can specify one of two keywords:

- Continue** resumes execution on the command *following* the one that caused the exception, staying within the same scope as the trap.
- Break** exits the scope that contains the trap, passing the original exception to the parent scope for handling.

For example, consider this short script:

```
trap {  
    write-host "Exception trapped in the script"  
    -fore green -back black  
    continue  
}  
  
function test {  
    trap {  
        write-host "Exception trapped in the function"  
        -fore yellow -back black  
        break  
    }  
    write-host "I am inside the function"  
    get-wmiobject win32_process -comp NotOnline -ea stop
```

```
        write-host "I am still inside the function"  
    }  
  
    write-host "Running the function"  
    test  
    write-host "Finished running the function"
```

The exception occurs on the Get-WmiObject cmdlet. Because a trap is defined within the function's scope, that trap executes. The trap ends in break, so the shell exits the function's scope, passing the exception to the parent scope. A trap is defined there as well, and that trap executes. That trap ends in continue, so the shell resumes execution on the line following the one that caused the exception. From the script's perspective, it was the test function that caused the exception, so execution resumes on the line after that. The output from this script looks like this:

```
Running the function  
I am inside the function  
Exception trapped in the function  
Exception trapped in the script  
Finished running the function
```



### Follow this example and make sure you understand why its output is as shown.

Here is the same script again, with one minor change: The trap inside the function now ends with continue instead of break:

```
trap {  
    write-host "Exception trapped in the script"  
    -fore green -back black  
    continue  
}  
  
function test {  
    trap {  
        write-host "Exception trapped in the function"  
        -fore yellow -back black  
        continue  
    }  
    write-host "I am inside the function"  
    get-wmiobject win32_process -comp NotOnline -ea stop  
    write-host "I am still inside the function"  
}  
  
write-host "Running the function"  
test  
write-host "Finished running the function"
```

The output this time is:

```
Running the function  
I am inside the function  
Exception trapped in the function  
I am still inside the function  
Finished running the function
```

Do you see why? When the exception occurred, the trap within the function executed. However, it ended in continue, so the shell stayed within the function's scope and resumed execution on the line following the one that caused the exception. That's why *I am still inside the function* displayed this time. The

MCT USE ONLY. STUDENT USE PROHIBITED

function ended normally, by reaching its end, and so the script never perceived an exception. That's why the script's trap was never executed.

Following the scope tree is extremely important when using trap constructs. It can be complicated, but by taking the time to think about what the shell is doing, you can help avoid confusion.

MCT USE ONLY. STUDENT USE PROHIBITED

## Trap Notes

- The last slide's trap was a general trap, meaning it will execute for any kind of exception
- Traps can also be defined for specific types of exceptions
  - Creating a specific trap requires knowing its underlying .NET Framework class type

```
trap  
[System.Management.Automation.CommandNotFoundException]  
{"Command error trapped"}
```

- The preceding example executes only for exceptions of the System.Management.Automation.CommandNotFoundException type
  - This separation provides a way to define different error handling behavior for different types of problems
  - However, determining the correct class type can be difficult

## Key Points

The trap construct you just saw was a *general trap*, meaning it executes for any kind of exception. Windows PowerShell also enables you to define traps for specific types of exceptions, although you need to know the underlying .NET Framework class type of the exception. For example:

```
trap [System.Management.Automation.CommandNotFoundException]  
{"Command error trapped"}
```

This trap executes only for exceptions of the System.Management.Automation.CommandNotFoundException type. The shell permits you to define multiple trap constructs, each of which can handle a different kind of exception. This is one way to define a different error handling behavior for different types of problems. However, it can be difficult to determine the proper .NET Framework class name for exceptions.

Also note that a trap is its own scope, much like a function. A trap can access variables from outside itself, using normal rules for accessing variables. However, any variables a trap creates or sets are normally private to the trap itself. If you need a trap to change a variable that exists in its parent scope, use the Set-Variable cmdlet, along with its –scope parameter, to modify the desired variable.

## Demonstration: Using Trap

- Learn how to use the trap construct



### Key Points

In this demonstration, you will learn how to use the trap construct.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Trap.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Try Constructs

- A Try...Catch construct is more granular than a trap construct and can be easier to use.

```
try {  
    gwmi win32_service -comp notonline -ea stop  
} catch {  
    write-host "Error!"  
} finally {  
    write-host "This executes either way"  
}
```

- Three components exist for this construct:

- The **Try** block defines the commands that you think might cause an error. These are the commands for which you've specified an -ErrorAction of Stop.
- If an exception occurs, the code inside the **Catch** block is executed.
- A **Finally** block will execute whether an exception occurred or not.

## Key Points

A Try...Catch construct is more granular than a trap construct, and can be easier to use. A typical construct might look like this:

```
try {  
    gwmi win32_service -comp notonline -ea stop  
} catch {  
    write-host "Error!"  
}
```

The Try block defines the commands that you think might cause an error, and for which you have specified an ErrorAction of Stop. If an exception occurs, the code inside the Catch block is executed.

**Note:** As with Trap constructs, you can specify multiple Catch blocks that each catch a specific type of exception. Run Help about\_try\_catch\_finally to learn more about this technique, if you need to.

After the code inside the Catch block finishes, execution resumes with whatever code follows the Try...Catch construct.

You can also specify a Finally block:

```
try {  
    gwmi win32_service -comp notonline -ea stop  
} catch {  
    write-host "Error!"  
} finally {  
    write-host "This executes either way"  
}
```

The code within the Finally block executes whether an exception occurred or not. You might use this block to close a database connection, for example, to ensure that the connection is closed whether an exception occurred or not.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Try

- Learn how to use the try construct



### Key Points

In this demonstration, you will learn how to use the Try construct.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Try.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## What's the Error?

- Errors within a Trap or Try construct are addressed via a built-in **\$Error** variable
  - `$Error[0]` is the most-recent error
  - `$Error[1]` is the next most-recent error
  - and so on
- The common parameter `-ErrorVariable` (aliased to `-EV`) can also be used to capture an error into a specified variable
  - In the code sample that follows, `-EV` captures the error into the variable `myerr`
  - Notice that `myerr` should not start with a `$`

```
try {  
    gwmi win32_service -comp notonline -ea stop -ev myerr  
} catch {  
    $myerr | out-file c:\errors.txt -append  
}
```

### Key Points

Within a Trap or Try construct, you may have need to access the exception that caused the construct to execute. There are two ways to do so.

A built-in `$error` variable contains the errors that have occurred in the shell. `$Error[0]` is the most recent error, `$Error[1]` the next most recent, and so on.

You can also use another common parameter, `-ErrorVariable` (or `-EV`), to capture the exception generated by that cmdlet into a variable. For example:

```
Get-Content does-not-exist.txt -EA Stop -EV myerr
```

Note that the variable name `myerr` is not preceded with a `$` when used in this context. If an exception occurs, the cmdlet places the exception information into the specified variable. You might use this as follows:

```
try {  
    gwmi win32_service -comp notonline -ea stop -ev myerr  
} catch {  
    $myerr | out-file c:\errors.txt -append  
}
```

Notice how `myerr` does not include a dollar sign when named in the `-EV` parameter because you are just specifying the variable's *name* at that point, and the dollar sign is not technically a part of the name. Later, when you actually *use* the variable, the dollar sign precedes the name to inform the shell that you are referencing a variable name.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using -ErrorVariable

- Learn how to use the –ErrorVariable parameter



### Key Points

In this demonstration, you will learn how to use the –ErrorVariable parameter.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using ErrorVariable.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Discussion: Combination Techniques

- What other combinations of Try and Trap might you consider using?
- In what situations?



### Key Points

Trap and Try are not mutually exclusive, and it is permissible to use both within the same script, function, or other element. You might use Try to trap errors for a specific cmdlet, and still have a top-level Trap construct to catch other, general exceptions that might occur.

What other combinations of Try and Trap might you consider using? In what situations?

## Lab C: Error Trapping and Handling

- Exercise 1: Retrieving Error Information
- Exercise 2: Handling Errors
- Exercise 3: Integrating Error Handling

Logon information

Virtual machine	LON-DC1	LON-SVR1
Logon user name	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

You are assigned to update a Windows PowerShell script that gathers audit information from machines on your network. All errors must be logged to the Windows Event Log under the Windows PowerShell event log according to company policy. This script is designed to run from the task scheduler, so the event log is the only way that you are notified if the script is not completing its audit as necessary.

#### Lab Setup

Before you begin the lab you must:

1. Start the **LON-DC1** virtual machines Wait until the boot process is complete.
2. Start the **LON-SVR1** virtual machine and log on with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. On LON-SVR1, open the Windows PowerShell ISE. All PowerShell commands are run within the Windows PowerShell ISE program.
4. In the PowerShell window, type the following command and press ENTER:  
`Set-ExecutionPolicy RemoteSigned -Scope CurrentUser.`
5. In the PowerShell window, type the following command and press ENTER:  
`Set-Location E:\Mod11\Labfiles`

## Exercise 1: Retrieving Error Information

### Scenario

Since your audit scripts need to run unattended, you would like to understand how Windows PowerShell commands respond when they encounter a problem. After seeing how Windows PowerShell acts on encountering an error, you need to find out how to control those errors and identify what problems were encountered. Finally, you want to be able to create your own errors, so you can provide meaningful messages for your environment.

1. Identify the Error Action Preference settings.
2. Control error action in scope.
3. Control error action per command.
4. Review errors globally.
5. Review errors by command.

#### ► Task 1: Identify the Error Action Preference settings

1. Examine the **\$ErrorActionPreference** variable.
2. Review the help for the preference variables and common parameters.

#### ► Task 2: Control nonterminating error action in scope

1. Confirm that **\$ErrorActionPreference** is set to **Continue**.
2. Run **ErrorActionPreference.ps1** and set the **FilePath** parameter to **Servers.txt** (the script and the text file are in the E:\Mod11\Labfiles folder).
3. Rerun the **ErrorActionPreference.ps1** script and set the **FilePath** parameter to **DoesNotExist.txt**.
4. Set **\$ErrorActionPreference** to its other values and run the script with a **FilePath** of **DoesNotExist.txt** for each value. Note the difference in how the error condition is handled.
5. Close Windows PowerShell.

#### ► Task 3: Control error action per command

1. Open **ErrorActionPreference.ps1** (from the E:\Mod11\Labfiles folder) in the Windows PowerShell ISE.
2. Confirm that **\$ErrorActionPreference** is set to **Continue**.
3. Use the **-ErrorAction** parameter for **Get-Content** (line 6) to control the error action at the cmdlet. Run the script once with a **FilePath** of **DoesNotExist.txt** for each error action setting.
4. Remove the **-ErrorAction** parameter from line 6 and save the file.

#### ► Task 4: Review errors globally

1. Open Windows PowerShell.
2. Confirm that **\$ErrorActionPreference** is set to **Continue**.
3. Run **ErrorActionPreference.ps1** and set the **FilePath** parameter to '**ServersWithErrors.txt**' (the text file is in the E:\Mod11\Labfiles folder).
4. Examine the **\$error automatic** variable.

► **Task 5: Review errors by command**

1. In the script pane of Windows PowerShell ISE, edit line **12** of the **ErrorActionPreference.ps1** script to include the **-ErrorVariable global:ComputerSystemError**.
2. Run **ErrorActionPreference.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt**.
3. Examine the **ComputerSystemError** variable.
4. Remove the **ComputerSystemError** variable from the global scope.
5. In the script pane of Windows PowerShell ISE, edit line **12** of the **ErrorActionPreference.ps1** script to include the **-ErrorVariable +global:ComputerSystemError**.
6. Run **ErrorActionPreference.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt**.
7. Examine the **ComputerSystemError** variable.
8. Remove the **-ErrorVariable +global:ComputerSystemError** from line **12** and save the script.

**Results:** After this exercise, you should have identified terminating and nonterminating errors, controlled the Windows PowerShell runtime's Error Action Preference both in scope and by command, examined where error information is available.

## Exercise 2: Handling Errors

Now that you have a feel for how errors occur and how to control their output, you need to identify the trouble spots and add some error handling to your scripts. In your experimentation, you notice that there are specific types of errors thrown and you need to identify how to deal with different types of errors.

The main tasks for this exercise are as follows:

1. Handle scoped errors.
2. Handle error prone sections of code.
3. Clean up errors in code.
4. Catch specific types of errors.

### ► Task 1: Handle scoped errors

1. Open the Windows PowerShell ISE and open **UsingTrap.ps1** (from the E:\Mod11\Labfiles folder).
2. Run **UsingTrap.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt** (from the E:\Mod11\Labfiles folder).
3. Modify the **trap** statement to display other information from the various errors.
4. Move the **trap** statement to different spots in the scope of the script (into the function **InventoryServer**, just under the first **if** statement) and see how that impacts the continuation after the error.
5. Remove the **-ErrorAction** parameters from the **Get-WmiObject** commands and run the script. Notice the change in behavior when the errors change from terminating errors to nonterminating errors.

### ► Task 2: Handle error prone sections of code

1. Open **AddingTryCatch.ps1** (from the E:\Mod11\Labfiles folder) in the Windows PowerShell ISE.
2. Change the **if** statement on lines 42 to 45 to be:

```
try {
    if ( $ping.Send("$server").Status -eq 'Success' )
    {
        InventoryServer($server)
    }
}
catch
{
    Write-Error "Unable to ping $server"
    Write-Host "The error is located on line number:
($_.InvocationInfo.ScriptLineNumber)"
    Write-Host "$($_.InvocationInfo.Line.Trim())"
}
```

3. Run the **AddingTryCatch.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt** (from the E:\Mod11\Labfiles folder).

### ► Task 3: Clean up after error prone code

- In the Windows PowerShell ISE, in the **AddingTry.ps1** file, after the closing brace in the **catch** add:

```
finally {
    Write-Host "Finished with $server"
}
```

**Question:** What type of scenarios would a **finally** block be useful for?

► **Task 4: Catch specific types of errors**

1. In the Windows PowerShell ISE, in the **AddingTry.ps1** file change the **catch** statement from:

```
catch {
```

to:

```
catch [System.InvalidOperationException] {
```

2. Run **AddingTry.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt**.
3. Open **UsingTrap.ps1** (from the E:\Mod11\Labfiles folder) in the Windows PowerShell ISE.
4. Change the **trap** statement from:

```
trap {
```

to:

```
trap [System.InvalidOperationException] {
```

5. Run **UsingTrap.ps1** and set the **FilePath** parameter to **ServersWithErrors.txt**.
6. Save both files and close the Windows PowerShell ISE.

**Results:** After this exercise, you should have an understanding of how trap and try/catch/finally can catch errors while a script is running. You should also have started to identify potential trouble spots in scripts as areas to focus your error handling efforts.

## Exercise 3: Integrating Error Handling

Having gotten comfortable with the error handling mechanisms in PowerShell, you need to incorporate that into your existing scripts. Because company policy dictates that errors are to be logged to the event log, you also need to determine how to write error log entries.

The main tasks for this exercise are as follows:

1. Set up the shell environment.
2. Create new Event Source for the Windows PowerShell Event Log.
3. Write error messages to the Windows PowerShell Event Log.
4. Add a top level Trap function to catch unexpected errors and log them to the event log.
5. Add localized Try/Catch/Finally error handling to potential trouble spots, and log the errors to the event log.

### ► Task 1: Set up the shell environment

1. On the **Start** menu click **All Programs**, click **Accessories**, click **Windows PowerShell**, right-click **Windows PowerShell** and choose **Run As Administrator**.
2. In the PowerShell window, type the following command and press ENTER:  
**Set-ExecutionPolicy 'RemoteSigned' -Scope CurrentUser**
3. In the PowerShell window, type the following command and press ENTER:  
**Set-Location E:\Mod11\Labfiles**

### ► Task 2: Create new Event Source for the Windows PowerShell Event Log

1. Start Windows PowerShell as an administrator.
2. Use **New-EventLog** to create an event source called **AuditScript** in the Windows PowerShell event log.

### ► Task 3: Write error messages to the Windows PowerShell Event Log

1. Use **Write-EventLog** to write a new event to the Windows PowerShell event log.
2. Open the event viewer and verify that the event was written to the event log.
3. Close Windows PowerShell.

### ► Task 4: Add a top level Trap function to catch unexpected errors and log them to the event log

1. Open the Windows PowerShell ISE and open **AuditScript.ps1**.
2. Create a **trap** function in the primary scope of the application to catch any unhandled errors and log them to the event log. The script name, type of error, the command generating the error, and the line number where the error occurred should all be logged to the event log.

### ► Task 5: Add localized Try...Catch error handling to potential trouble spots and log the errors to the event log

1. Review the **AuditScript.ps1** file and look for potential trouble spots. Wrap those spots in a **try/catch** or **try/catch/finally** block. Add other PowerShell statements as necessary to make the script hardened to dealing with errors.
2. Run **AuditScript.ps1** with the **FilePath** parameter to **AuditTest0.txt**, then **AuditTest1.txt**, and finally **AuditTest2.txt**.

3. Check the event log for and verify that any errors were logged. Also, compare the output files to verify that the outputs are all the same by using the commands that follow:

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest0.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest1.txt.xml)
```

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest1.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest2.txt.xml)
```

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest0.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest2.txt.xml)
```

4. Close the Windows PowerShell ISE.

**Results:** After this exercise, you should have created an event source in the Windows PowerShell event log, created a top-level trap to catch unexpected errors and log them, and wrapped potentially error-prone sections of script in a try/catch block.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. How can you find out what members the error records have?
2. What type of information are the error records providing you?
3. What type of tasks might you need to do regardless of whether a task succeeds or fails?
4. How would you find any cmdlets that work with the event log?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 5

# Debugging Techniques

- Explain the goal and practices for debugging
- Add debug trace messages to a script
- Enable and disable debug trace messages in a script
- Use the interactive step debugger to follow a script's execution
- Set, use, and remove breakpoints to debug a script
- Use the Windows PowerShell ISE to debug a script



Logic errors are perhaps the most difficult ones to correct. You usually do not anticipate them, and they do not always result in a clear error message. Instead, they result in your script not behaving the way you intended. The purpose of debugging is to discover the cause of logic errors, to resolve them, and to test the resolution.

### Lesson Objectives

After completing this lesson, you will be able to:

- Explain the goal and practices for debugging.
- Add debug trace messages to a script.
- Enable and disable debug trace messages in a script.
- Use the interactive step debugger to follow a script's execution.
- Set, use, and remove breakpoints to debug a script.
- Use the Windows PowerShell ISE to debug a script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Debugging Goal

- Recall: The goal of debugging is to resolve logic errors
  - The most common cause of logic errors is a variable or object property that contains a value other than what is expected or anticipated
  - Debugging is designed to help you see what those variables and properties actually contain
- However, to begin debugging, you must have an expectation of what your script will do, on a line-by-line basis
  - Make notes of input values
  - Walk through scripts, line by line
  - Make notes of what each line should do
  - Document values that variables should contain

### Key Points

As mentioned in the previous lesson, debugging is designed to resolve logic errors. The most common cause of logic errors is a variable or object property that contains a value other than the value you expected or anticipated. Debugging, then, is designed to help you see what variables and properties actually contain, so that you can revise both your expectations and your script accordingly.

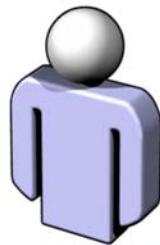
However, to begin debugging, you *must* have an expectation of what your script will do, line by line. You won't know whether your expectations are erroneous unless you have expectations to begin with.

Before you even begin debugging, you should sit down with your script and a piece of paper. Make a note of what input values, if any, your script is to use. Walk through the script, line by line. Make a note of what you think each line will do and how you would verify each line's correct operation. Write down the value that each variable will contain, and note the values that you believe will be contained in object properties.

**Note:** As you become more experienced, you will not need to actually use a piece of paper. Until then, however, writing this information down will save time and effort.

## Discussion: Setting Expectations

- As a group, review the script in your Student Guide
- Discuss what the script will do
  - Make note of what object properties and variables should contain
  - Write everything on a piece of paper or a white board



### Key Points

Review the following script as a group. Discuss what each line of the script will do, and make a note of what object properties and variables should contain. Write everything on a piece of paper or a white board.

```
param ( [string]$computername = (read-host "Enter the computer to query"))

# PROBLEMS:
# - Doesn't display the correct total count of disks
# - Doesn't display any hard/fixed disk information
# - When you get it to display fixed disk info, it
#   doesn't display total volume size
# NOTE:
# - Doesn't display any errors if you provide a valid computer name

# retrieve all logical disks
$disks = get-wmiobject win32_logicaldisk -computername $computername

# start with zero disks
$diskcount = 0

# go through all disks
foreach ($disk in $disks) {

    # add to count
    $diskcount++

    # for hard drives, display info
    if ($disk drivetype -eq 'fixed') {
        $disk | select deviceid,space,freespace
    }
}
```

```
# display total disks
write-host "$diskcount total disks on $computername, only fixed disks shown"
```

**Note:** You will see this script many times throughout this lesson. Our goal isn't to discover and fix the bugs, but rather to use this as an example for practicing different debugging techniques.

MCT USE ONLY. STUDENT USE PROHIBITED

## Trace Messages

- Trace messages help you follow the flow of script execution. They help you determine what values are actually within object properties and variables.
  - PowerShell does not provide automated trace messages.
  - You add these to your script by using the Write-Debug cmdlet.

```
$var = Read-Host "Enter a computer name"
Write-Debug "`$var contains $var"
```

- Write-Debug output is suppressed by default. Enable its output by modifying the \$DebugPreference variable at the beginning of your script.

```
$DebugPreference = 'Continue'
$DebugPreference = 'SilentlyContinue'
```

- Debug output is colored differently from normal output. Some IDEs can redirect debug output to a different pane.

### Key Points

One debugging technique is to have your script output *trace messages*, which help you follow the flow of the script's execution and help you determine what values are actually within object properties and variables. When you compare the trace messages to your written expectations for the script, everything should match. If something does not match, then you have probably found your bug.

The shell does not provide automated trace messages. Rather, you add these to your script by using the Write-Debug cmdlet. For example, each time you change the value of a variable, you should write the new value:

```
$var = Read-Host "Enter a computer name"
Write-Debug "`$var contains $var"
```

**Note:** Notice that this Write-Debug example uses double quotation marks, which will replace variables with their contents. The first instance of the variable, however, has the dollar sign escaped. That will result in the variable name being displayed as-is, followed by *contains* and the variables value. This debug output is useful because it lists the variable name *and* its value.

You should also include Write-Debug whenever your script makes a decision. For example, consider the following:

```
If ($var -notlike "*srv*") {
    Write-Host "Computer name is not a server"
}
```

You might modify this as follows:

```
If ($var -notlike "*srv*") {
```

```
    Write-Debug "$var does not contain 'srv'"
    Write-Host "Computer name $var is not a server"
} else {
    Write-Debug "$var contains 'srv'"
}
```

Notice that an Else block has been added so that the script generates trace output no matter which way the logical decision turns out.

By default, the shell suppresses output from Write-Debug. To enable this output, modify the built-in \$DebugPreference variable at the beginning of your script:

```
$DebugPreference = 'Continue'
```

Doing so enables Write-Debug output. When you are finished debugging your script, you do not need to remove the Write-Debug commands. Instead, suppress their output again:

```
$DebugPreference = 'SilentlyContinue'
```

This allows you to leave the Write-Debug commands in place in case they are needed for future debugging efforts.

**Note:** Write-Debug output is colored differently from normal output in the Windows PowerShell console window. Third-party script editors may redirect this output to a different pane, window, or tab, helping to separate your script's normal output from the trace messages.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Write-Debug

- Learn how to use the Write-Debug



### Key Points

In this demonstration, you will learn how to use Write-Debug.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Write-Debug.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Step Debugger

- Reviewing hundreds of lines in a large script for trace messages can be time-consuming and difficult
- The shell's step debugger enables script execution to occur one line at a time. Before executing each script line, you can choose to
  - Pause (suspend) the script
  - Review the contents of variables and object properties
  - Execute the next line
- Two commands start and stop the step debugger:

```
Set-PSDebug -step  
Set-PSDebug -off
```

### Key Points

For a lengthy script, reviewing hundreds of lines of trace messages can be time-consuming and difficult. In some cases, you may instead prefer to use the shell's step debugger. The step debugger enables you to execute your script one line at a time. Before executing a line of your script, you can choose to pause, or *suspend*, the script, and review the contents of variables, object properties, and so forth.

Enable the step debugger by running:

```
Set-PSDebug -step
```

When you are finished using it, disable the step debugger by running:

```
Set-PSDebug -off
```

When your script is executing, you can tell the step debugger to go ahead and execute a line, or you can ask it to suspend the script. During suspension, the shell displays a different prompt, which helps you remember that you are inside your script. You can execute commands normally, including displaying objects and variables. Run Exit to resume script execution.

## Demonstration: Using the Step Debugger

- Learn how to use the step debugger



### Key Points

In this demonstration, you will learn how to use the step debugger.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using the Step Debugger.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Breakpoints

- Stepping through large scripts can also be time-consuming with the step debugger
  - Clicking 200 times to move through a 200-line script takes a lot of time
- Breakpoints allow you to preset locations in a script where execution should suspend. Breakpoints are managed using a set of cmdlets:
  - **Set-PSBreakpoint** creates a new breakpoint condition
  - **Remove-PSBreakpoint** deletes a breakpoint condition
  - **Disable-PSBreakpoint** stops using a breakpoint condition, but leave it in place
  - **Enable-PSBreakpoint** enables a disabled breakpoint condition
  - **Get-PSBreakpoint** retrieves one or more breakpoint conditions

### Key Points

One disadvantage of the step debugger is that it does not provide you with a means to skip over portions of your script that you know are working. If you have a 200-line script, and you know you have a problem near the end, having to press "Yes" a hundred or so times just to get to the buggy portion of your script can be tedious.

Windows PowerShell offers *breakpoints* to help alleviate that situation. With a breakpoint, you run your script normally, rather than using the step debugger. However, when the shell encounters a breakpoint condition that you have set, it automatically suspends the script much as the step debugger does. You can then examine property values or variables, and run Exit to resume script execution.

You can set the following breakpoint conditions:

- Suspend when the script reaches a certain line
- Suspend when a certain variable is read
- Suspend when a certain variable is changed, or written
- Suspend when a certain variable is read or written
- Suspend when a certain command or function is executed

You can also specify a particular action, which is a set of Windows PowerShell commands, that you want to run in response to the breakpoint.

Breakpoints are managed by using cmdlets:

- Set-PSBreakpoint—create a new breakpoint condition
- Remove-PSBreakpoint—delete a breakpoint condition
- Disable-PSBreakpoint—stop using a breakpoint condition, but leave it in place

- `Enable-PSBreakpoint`—enable a disabled breakpoint condition
- `Get-PSBreakpoint`—retrieve one or more breakpoint conditions

Breakpoints allow you to define the conditions under which your script suspends (or takes some other action).

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Using Breakpoints

- Learn how to use breakpoints



### Key Points

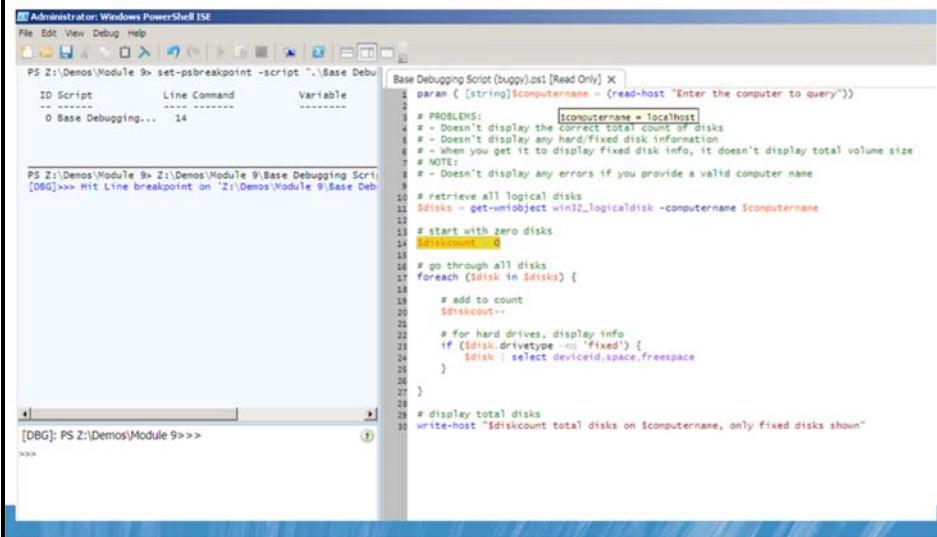
In this demonstration, you will learn how to use breakpoints.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Using Breakpoints.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Debugging in the ISE

- The ISE includes basic visual support for debugging, primarily in conjunction with breakpoints



```

Administrator: Windows PowerShell ISE
File Edit View Debug Help
PS Z:\Demos\Module 9> set-psbreakpoint -script ".\Base Debug Script.ps1" -line 14
ID Script Line Command Variable
----- -----
0 Base Debugging... 14

PS Z:\Demos\Module 9> Z:\Demos\Module 9\Base Debug Script.ps1
[DBG]: PS Z:\Demos\Module 9>>>

```

```

Base Debugging Script (buggy).ps1 [Read Only] X
param ([string]$computername = <read-host "Enter the computer to query">)
# PROBLEMS:
# - Doesn't display the correct total count of disks
# - Doesn't display any hard/fixed disk information
# - When you get it to display fixed disk info, it doesn't display total volume size
# NOTE:
# - Doesn't display any errors if you provide a valid computer name

# retrieve all logical disks
$disks = get-wmobject win32_logicaldisk -computername $computername
# start with zero disks
$diskcount = 0
# go through all disks
foreach ($disk in $disks) {
    # add to count
    $diskcount++
    # for hard drives, display info
    if ($disk drivetype -eq 'fixed') {
        $disk | select deviceid,space,freespace
    }
}

# display total disks
write-host "$diskcount total disks on $computername, only fixed disks shown"

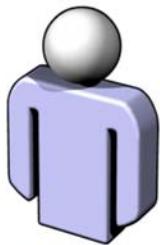
```

### Key Points

The Windows PowerShell debugger provides basic visual support for debugging, primarily in conjunction with breakpoints. There is no visual means for creating a breakpoint. However, once you create a line number breakpoint for a script (using the `-script` parameter of `Set-PSBreakpoint`), the ISE displays the breakpoint within the script by highlighting the specified line. When you run that script, the ISE enables you to hover your mouse pointer over variable names to see the contents of those variables.

## Discussion: Which Technique Is Best?

- You will probably find yourself using a mix of debugging techniques, depending on the exact situation
- What are the situations where these techniques can work best?
  - Step debugger
  - Breakpoints
  - Write-Debug
  - Combinations?



### Key Points

You will probably find yourself using a mix of debugging techniques depending on the exact situation. The step debugger can be useful for short scripts; breakpoints are more flexible, especially in longer scripts, but require you to run additional commands to create and manage them. Write-Debug can be useful in the step debugger or in combination with breakpoints.

## Lab D: Debugging a Script

- Exercise 1: Debugging from the Windows PowerShell Console
- Exercise 2: Debugging Using the Windows PowerShell Integrated Script Environment (ISE)

Logon information

Virtual machine	LON-DC1
Logon user name	Contoso\Administrator
Password	Pa\$\$w0rd

**Estimated time: 30 minutes**

### Estimated time: 30 minutes

Your manager has provided you with several Windows PowerShell scripts that are not functioning as expected. The first script is EvaluatePingTimes, which loads a log of ping tests run against various machines on your WAN. It categorizes the results based on the expected network latency. However, there is a problem with the totals not matching the number of ping tests performed. The second script is CalculatePerfMonStats, which imports some PerfMon data relating to memory usage on a server and calculates some statistics. For some of the counters, there seems to be a problem with how the numbers are read in, as some math errors are generated.

### Lab Setup

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

1. Start the **LON-DC1** virtual machine and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open a Windows PowerShell session.

## Exercise 1: Debugging from the Windows PowerShell Console

### Scenario

Since you are responsible for fixing several error prone scripts, you need to experiment with some of the debugging facilities in PowerShell. First, you explore how debugging statements can be written to the console. After that, you can explore how PowerShell allows you to step through the script and interactively explore the current state of variables. Finally, after finding some of your problem areas, you can use the breakpoint features to stop a script at the trouble spots for further investigation.

The main tasks for this exercise are as follows:

1. Use \$DebugPreference to enable debugging statements.
2. Use Write-Debug to provide information as the script progresses.
3. Use Set-PSDebug to trace the progress through the script.
4. Use Set-PSBreakpoint to break into a script's progress at specified points.
5. Use Get-PSBreakpoint and Remove-PSBreakpoint to manage breakpoints.
6. Use Set-PSBreakpoint to break into a script if certain conditions are met.

#### ► Task 1: Use \$DebugPreference to enable debugging statements

1. Run the **EvaluatePingTimes.ps1** script and pass the **PingTestResults.xml** file as the **-Path** parameter and use the **-FirstRun** switch (the script and the xml file are in the E:\Mod11\Labfiles folder).
2. Set the **\$DebugPreference** variable to **Continue**.
3. Re-run the **EvaluatePingTimes.ps1** script with the same parameters.

#### ► Task 2: Use Write-Debug to provide information as the script progresses

1. Open **CalculatePerfMonStats.ps1** (the script is in the E:\Mod11\Labfiles folder) in Notepad.
2. Add **Write-Debug** statements inside functions, inside loops, and in other areas where variables change or are assigned.
3. In the Windows PowerShell prompt, verify that **\$DebugPreference** is set to **Continue**.
4. Run the **CalculatePerfMonStats.ps1** with a **-Path** parameter of **PerfmonData.csv** (which can be found in the E:\Mod11\Labfiles folder).

#### ► Task 3: Use Set-PSDebug to trace the progress through the script

1. Set **\$DebugPreference** to **SilentlyContinue**.
2. Use **Set-PSDebug** to set the trace level to **1**.
3. Run the **EvaluatePingTimes.ps1** script and pass the **PingTestResults.xml** file as the **-Path** parameter and use the **-FirstRun** switch.
4. Use **Set-PSDebug** to set the trace level to **2** and rerun the **EvaluatePingTimes.ps1** script with the same parameters as before.
5. Use **Set-PSDebug** to step through the script.
6. Turn off script debugging.

#### ► Task 4: Use Set-PSBreakpoint to break into a script's progress at specified points

1. Use **Set-PSBreakpoint** to set breakpoints at lines **36, 40, 44, and 48** of **EvaluatePingTimes.ps1**.
2. Run the script and pass the **PingTestResults.xml** file as the **-Path** parameter and use the **-FirstRun** switch.
3. Explore the values in the script at the point in time. (Remember - **\$\_** is the current object in the pipeline or the current object being evaluated in a switch statement!)

- ▶ **Task 5: Use Get-PSBreakpoint and Remove-PSBreakpoint to manage breakpoints**
  - Remove the breakpoints from lines **36**, **40**, and **48**.
- ▶ **Task 6: Use Set-PSBreakpoint to break into a script if certain conditions are met**
  1. Remove all the existing breakpoints.
  2. Create a breakpoint on line **44** of **EvaluatePingTimes.ps1** and use the **-Action** parameter to check if the **RoundtripTime** property is equal to **35**. If it is, have the script break into a nested prompt. If not, let the script continue.
  3. Run the script and pass the **PingTestResults.xml** file as the **-Path** parameter and use the **-FirstRun** switch.
  4. Close the Windows PowerShell prompt.

**Results:** After this exercise, you should have an understanding of how the DebugPreference automatic variable controls debugging messages, know how to step through troublesome scripts and track how things are changing, and set breakpoints in a script to further troubleshoot script performance.

## Exercise 2: Debugging Using the Windows PowerShell Integrated Scripting Environment (ISE)

### Scenario

Having just finished using the debugging tools from the console, you want to take advantage of the graphical script editor the Windows PowerShell ISE provides. You will use the debugging skills developed in the previous exercise to step through and work with another problem script.

The main tasks for this exercise are as follows:

1. Set breakpoints directly in the script editor.
2. Step through a script using the menu shortcuts or the keyboard.

#### ► Task 1: Set a breakpoint

1. In the script editor, set your cursor on line **33**. Use the **Toggle Breakpoint** function from the **Debug** menu to set a breakpoint.
2. Right click on line **25** and choose **Toggle Breakpoint**.
3. Set your cursor on line **44** and use F9 to set a breakpoint.
4. In the command window, use **Get-PSBreakpoint** to examine breakpoints you have set.
5. Remove those breakpoints.

#### ► Task 2: Step through the script

1. Set a breakpoint on the first statement of the **foreach** loop where **CreateStatistics** is called and which breaks in only if **\$CounterValue** is **Memory\Pages/sec**.

**Hint:** Check the help on **Set-PSBreakpoint** for how to set a breakpoint on a command.

2. From the command window, run the script with the **-Path** parameter set to the **PerfmonData.csv** file (in the E:\Mod11\Labfiles folder).
3. When the script breaks, use F11 or the **Step Into** option off of the **Debug** menu to step into the **CreateStatistics** function. Step into the function one more time.
4. Step over (using the keyboard or the Debug menu) the first line in the **CalculateMedian** function.
5. Step out of the **CalculateMedian** function.
6. Move into the command window and examine the **\$Stats** variable.
7. Continue running the script.
8. Close the Windows PowerShell ISE.

**Results:** After this exercise, you should have used the tooling built in to the Windows PowerShell ISE to debug a script.

## Lab Review

1. What are the key variables in your script and when are they used?
2. Are your commands getting the correct inputs?
3. Are there hard-coded paths, file names, IP addresses, or other resources that may vary depending on where the script is run?
4. Are the variable names spelled correctly?
5. Are the variable names easy to distinguish from each other?

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 6

# Modularization

- Modularize one or more commands into a basic function
- Write a parameterized function
- Write a pipeline (filtering) function that outputs text
- Write a pipeline (filtering) function that outputs custom objects
- Explain the advantages and components of an advanced function



As you develop more complex and useful scripts, you will want to reuse certain portions again and again. Modularization is a technique for packaging portions of code into more easily reused units. Windows PowerShell provides several different levels of modularization, each of which grows successively more complex while providing an increasing level of flexibility and utility.

### Lesson Objectives

After completing this lesson, you will be able to:

- Modularize one or more commands into a basic function.
- Write a parameterized function.
- Write a pipeline (filtering) function that outputs text.
- Write a pipeline (filtering) function that outputs custom objects.
- Explain the advantages and components of an advanced function.

MCT USE ONLY. STUDENT USE PROHIBITED

## Why Modularize?

- Modularization is designed to make useful, reusable, self-contained components.
  - It makes scripting faster by enabling you to easily reuse portions of code from other projects.
- PowerShell's basic form of modularization is the function. Functions should generally:
  - Be as single-purpose and self-contained as possible.
  - Align to a real-world task.
  - Rely only on specific, defined input.
  - Never try to access information from outside itself.
  - Output data in a way that offers the maximum opportunity for reuse. Generally this means putting data in the pipeline.

### Key Points

Modularization is designed to make useful, reusable, self-contained components. The purpose of modularization is to make scripting faster by enabling you to reuse useful portions of code from other projects. In Windows PowerShell, the basic form of modularization is a *function*. You have seen functions briefly used elsewhere in this course, but in this lesson you will explore their full range of functionality.

Functions should generally be as single-purpose and as self-contained as possible. By having a function do one single task, you make it easier to reuse that function in places where that task is needed.

Tasks might include things like writing to a database, testing connectivity to a remote computer, validating a server name, and so forth. Ideally, functions should align themselves to one real-world task, in much the same way that most shell cmdlets do.

**Note:** You can think of cmdlets as another form of modularization, although true cmdlets require a knowledge of .NET Framework programming.

Functions should rely only on specific, defined input to accomplish their task. A function should never try to access information from outside of itself because doing so limits the ways in which a function can be reused. For example, suppose you have a script that contains a function named Get-ServerName. If the function relies on being able to read specific variables from its parent script, any script that uses that function must provide those specific variables. That dependency makes it more difficult to reuse the function in different scripts. However, by keeping the function self-contained, you can more easily reuse it in a broader variety of situations.

Functions should output data in a way that offers the maximum opportunity for reuse. Generally, this means placing output into the pipeline. For example, a function that outputs data directly to a CSV file can be used only when you want a CSV file as output. A function that outputs data to the pipeline,

however, can be used in a much broader variety of scenarios because you can pipe that output to Export-Csv, Export-Clixml, ConvertTo-HTML, or any other cmdlet, without having to change your function.

MCT USE ONLY. STUDENT USE PROHIBITED

## Discussion: What Would You Modularize?

- What sorts of tasks can you envision performing in many different scripts?
  - Testing remote computer connectivity
  - Testing remote computer availability
  - Validating data
  - What else?



### Key Points

What sorts of tasks can you envision performing in many different scripts?

- Testing remote computer connectivity
- Testing remote computer availability
- Validating data
- What else?

## Basic Functions

- A basic function accepts no input, which means it is capable of performing its task without needing any additional information
  - A basic function may produce output, which can be piped elsewhere
- To create the function:

```
Function Do-Something {  
    # function code goes here  
}
```

- To invoke the function:

```
Do-Something
```

### Key Points

A basic function is the simplest form of modularization. It accepts no input, which means it is capable of performing its task without needing any additional information. It may produce output, which could be piped to other cmdlets.

A basic function is declared with the keyword function, and the contents of the function are contained within curly braces:

```
Function Do-Something {  
    # function code goes here  
}
```

Functions must be defined before they can be used, which means you will often have your script's functions at the start of the script. To run a function that has been defined, use the function name, much as you would a cmdlet:

```
Do-Something
```

## Demonstration: Basic Functions

- Learn how to write a basic function



### Key Points

In this demonstration, you will learn how to write a basic function.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Basic Functions.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Function Output

- The **Write-Output** cmdlet
  - Is the proper way to output information from a function.
  - Is aliased to Write.

```
Write-Output $var  
Write $var  
$var
```

- The **Return** keyword
  - Is another mechanism for returning output, also to the pipeline. When complete, Return exits the function.

```
Return $var
```

- Note: Do not use Write-Host to output function data. It sends text to the console window and not the pipeline.

### Key Points

The Write-Output cmdlet is the proper way to output information from a function. This cmdlet, whose alias is Write, sends objects to the pipeline, where they can be re-used by other cmdlets.

There are actually three ways to use Write-Output. Suppose you have your output in a variable named \$var. All three of these examples are functionally identical:

```
Write-Output $var  
Write $var  
$var
```

In addition, functions can return output by using the Return keyword:

```
Return $var
```

Return is special in that it writes whatever you tell it to the pipeline, and then immediately exits the function.

*Do not* use Write-Host to output data from a function. Write-Host does *not* write information to the pipeline; instead, it sends text directly to the console window. That text cannot be piped to other functions or cmdlets, thus limiting the function's reusability.

## Parameterized Functions

- A parameterized function adds the ability to pass information into the function
- Two ways to define a function's input parameters:
  - As part of the function declaration

```
Function Do-Something
  ($computername,$domainname) {
    # function code goes here
}
```

- Through an inside-the-function Param() declaration

```
Function Do-Something {
  Param(
    $computername,
    $domainname
    )
    # function code goes here
}
```

### Key Points

A parameterized function takes a basic function and adds the ability to pass information into the function.

Typically, functions should not have any hard-coded values, such as for computer names or user names or whatever. Instead, functions should accept this information as input parameters, thus enabling the function to be used to a broader variety of situations.

There are two ways to define a function's input parameters. The first is as part of the function declaration:

```
Function Do-Something ($computername,$domainname) {
  # function code goes here
}
```

**Note:** Try to name your functions using the verb-singular\_noun naming convention that cmdlets use. Be careful not to use a function name that is the same as an existing cmdlet.

The second, and preferred way to define input parameters is as follows:

```
Function Do-Something {
  Param(
    $computername,
    $domainname
    )
  # function code goes here
}
```

This method is preferred because it is easier to read, and because it follows the convention used by the shell to store functions in memory. It is also the format you used to define parameters for a script. Note that the formatting is flexible; you could also do this:

```
Function Do-Something {  
    Param($computername,$domainname)  
    # Function code goes here  
}
```

Breaking each parameter out onto its own line can make the parameters easier to read. This is especially true if you follow best practices of defining an input data type and default values for your parameters:

```
Function Do-Something {  
    Param(  
        [string]$computername = 'localhost',  
        [string]$domainname = 'contoso.com'  
    )  
    # function code goes here  
}
```

As with parameterized scripts, you can also define parameters whose defaults prompt the user, or throw an error, if a value is not provided:

```
Function Do-Something {  
    Param(  
        [string]$computername = $(Read-Host 'Computer name'),  
        [string]$domainname = $($throw 'Domain name required.')  
    )  
    # function code goes here  
}
```

When you run a parameterized function, you can pass input values by position:

```
Do-Something localhost 'contoso.com'
```

Note that parameters are not separated by commas as in some other programming languages; functions, like cmdlets, separate parameters by using spaces. You can also pass input values by using parameter names, which enables you to put the parameters in any order:

```
Do-Something -domainname contoso -computername localhost
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Parameterized Functions

- Learn how to write a parameterized function



### Key Points

In this demonstration, you will learn how to write a parameterized function.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Parameterized Functions.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Pipeline Input

- Functions can also accept pipeline input
  - In a parameterized function the shell automatically places pipeline input into a special variable named \$input
  - You do not need to declare \$input as a parameter
  - You will have to enumerate through \$input, usually via ForEach-Object because it will often contain multiple objects

```
function Do-Something {  
    param (  
        $domain = 'contoso.com'  
    )  
    foreach ($computer in $input) {  
        write "Computer $computer, domain $domain"  
    }  
}  
  
'localhost','server1' | do-something -domain adatum
```

### Key Points

So far, you have seen how to pass input values to a function by means of parameters. Functions can also accept pipeline input. In a normal parameterized function, the shell automatically places pipeline input into a special variable named \$input. You do not need to declare \$input as a parameter. You do, however, have to enumerate through \$input because it often contains multiple objects. A ForEach construct is the best way to enumerate through the \$input collection. Here is an example:

```
function Do-Something {  
    param (  
        $domain = 'contoso.com'  
    )  
    foreach ($computer in $input) {  
        write "Computer $computer is in domain $domain"  
    }  
}  
  
'localhost','server1' | do-something -domain adatum
```

Note that the pipeline input of two computer names is placed into \$input, and "adatum" is placed into the standard-style parameter \$domain.

This is only one way in which a function can accept pipeline input. A somewhat more efficient and easier way is to change the function to be a *filtering function*, also called a *pipeline function*.

## Filtering Functions

- A filtering function is specially designed to accept pipeline input. It consists of three named script blocks:
  - **BEGIN** executes once when the function is first called. You can use this block to perform any setup tasks that the function needs, such as connecting to a database.
  - **PROCESS** executes one time for each object that was piped in from the pipeline. Within this block, the special `$_` placeholder contains the current pipeline object.
  - **END** executes once after all of the pipeline input objects have been processed. You can use this block to perform any clean-up tasks, such as closing a database connection.

```
function Do-Something {
    BEGIN {}
    PROCESS {}
    END {}
}
```

### Key Points

A filtering function is specially designed to accept pipeline input. The function consists of three named script blocks:

- BEGIN: This script block executes once when the function is first called. You can use this block to perform any setup tasks that the function needs, such as connecting to a database.
- PROCESS: This script block is executed one time for each object that was piped in from the pipeline. Within this block, the special `$_` placeholder contains the current pipeline object.
- END: This script block executes once after all of the pipeline input objects have been processed. You can use this block to perform any clean-up tasks, such as closing a database connection.

You do not have to supply all of these script blocks; include only the ones you need. Some administrators prefer to include all three blocks even if they are not all used, which is also fine. Here is an example:

```
function Do-Something {
    param (
        $domain = 'contoso.com'
    )
    BEGIN {}
    PROCESS {
        $computer = $_
        write "Computer $computer is in domain $domain"
    }
    END {}
}

'localhost','server1' | do-something -domain adatum
```

The PROCESS script block essentially acts as a built-in ForEach construct, automatically enumerating the pipeline input and placing each object into the `$_` placeholder in turn. Notice that normal parameters are also still supported.

## Demonstration: Filtering Functions

- Learn how to write a filtering function
- See how to use a standard input parameter in a filtering function



### Key Points

In this demonstration, you will learn how to write a filtering function. You will also see how to use a standard input parameter in a filtering function.

#### Demonstration steps:

1. Start virtual machines **LON-DC1**, **LON-SVR1** and **LON-SVR2** then log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Filtering Functions.ps1**, and **E:\Mod11\Democode\Filtering Functions 2.ps1**. You should start with **Filtering Functions.ps1** first.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Function Output: Text

- The filtering functions you've seen so far have run PowerShell commands, placing their output into the pipeline
  - By doing so, these commands' output becomes the function's output
  - Yet PowerShell doesn't always do a good job of displaying output when the pipeline contains multiple different types of objects
  - An alternate option is to construct your own output
  - For example, you might output text to form column headers, then write each row of output
  - PowerShell includes a formatting operator "-f" that makes it easier to create custom output

### Key Points

The filtering functions you have seen so far have run Windows PowerShell commands and allowed those commands to place their output into the pipeline. By doing so, those commands' output become the function's output. In the previous demonstration, for example, this included one or two calls to Get-WmiObject.

Windows PowerShell doesn't always do a good job of displaying output when the pipeline contains multiple different types of objects. For example, when the pipeline contained both a Win32\_OperatingSystem object and a Win32\_BIOS object, the output wasn't displayed very well.

One option for improving the situation would be to construct your own output. You might, for example, output text to form column headers, and then write each row of output. Windows PowerShell even includes a formatting operator, -f, that can make it easier to create this kind of output.

## Demonstration: Text Function Output

- Learn how to write a function that outputs formatted text
- Learn how to use the -f formatting parameter



### Key Points

In this demonstration, you will learn how to write a function that outputs formatted text. You will also learn to use the -f formatting operator.

### Demonstration steps:

1. Start virtual machines **LON-DC1**, **LON-SVR1**, and **LON-SVR2** then log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Text Function Output.ps1**, and **E:\Mod11\Democode\Text Function Output 2.ps1**. Start with **Text Function Output.ps1** first.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Why Text Output Is a Bad Choice

- Text (string) output is inherently difficult to reuse
  - It cannot be reformatted for table, CSV, XML, or other output using standard cmdlets
  - Ensuring output data remains as an object ensures that it can be more easily worked with in the future
- The **PSObject** object
  - Is a very simple, blank, custom object
  - Serves as a blank canvas to which you attach your own properties, and ultimately your own data
  - Enables you to keep otherwise-textual data in an object

```
$obj = New-Object PSObject
$obj | Add-Member NoteProperty ComputerName $computernode
Write $obj
```

### Key Points

The previous two examples output text, or more specifically String objects. The problem with this approach is that text is inherently difficult to re-use. For example, what if you wanted to output information to a CSV file instead of to an on-screen table? Using the previous two examples as a starting point, you would have to rewrite quite a bit of the function in order to have CSV output. If later you needed XML output, you would be rewriting the function again. HTML-formatted output? Rewrite.

As you have learned already, Windows PowerShell is an object-oriented shell, and it works best when you give it objects, rather than text, to work with.

The previous two examples retrieved two WMI classes: Win32\_OperatingSystem and Win32\_BIOS. Because these are two different classes, the shell cannot easily display them in a single line of a table—at least, not using the default formatting system. Our decision to use text was one approach to combining those two pieces of information onto a single line, but it's an approach that doesn't complement the way Windows PowerShell works.

Another way to combine that information is to create a blank, custom object:

```
$obj = New-Object PSObject
```

A "PSObject" is a very simple object that serves as a sort of blank canvas. You can then attach your own properties to it. For example, to add a ComputerName property:

```
$obj | Add-Member NoteProperty ComputerName $computernode
```

This adds a new property of the NoteProperty type. A NoteProperty is a static value attached to an object. The new NoteProperty is named ComputerName, and its value is whatever is in the \$computernode variable.

You can continue to attach properties until your custom object contains all of the information you need. Then, you output that custom object to the pipeline:

```
Write $obj
```

Because you are writing a single type of object to the pipeline, Windows PowerShell can format the output more easily. You can also pipe those objects to other cmdlets to export them to CSV, format them as HTML, and so forth. By using objects as function output, you enable easier reuse of the function's output in a broader variety of scenarios.

## Demonstration: Object Function Output

- Learn how to write a function that outputs objects



### Key Points

In this demonstration, you will learn how to write a function that outputs objects.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Object Function Output.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## More Than One Way to Do Everything

```
function Get-Inventory {
    PROCESS {
        $computer = $_
        $os = gwmi win32_operatingsystem -comp $computer
        $bios = gwmi win32_bios -comp $computer
        $obj = new-object psobject
        $obj | select
            @{Label='ComputerName';Expression={$computer}},
            @{Label='SPVersion';Expression={$os.servicepackmajorversion}},
            @{Label='BIOSSerial';Expression={$bios.serialnumber}},
            @{Label='BuildNo';Expression={$os.buildnumber}}
    }
    gc names.txt | get-inventory
}

gwmi win32_operatingsystem -computer (gc names.txt) |
    select @{Label='ComputerName';Expression={$_.__SERVER}},
        @{Label='BuildNo';Expression={$_._BuildNumber}},
        @{Label='SPVersion';Expression={$_._ServicePackMajorVersion}},
        @{Label='BIOSSerial';Expression={
            (gwmi win32_bios -comp $_.__SERVER).serialnumber
        }}}
```

### Key Points

As you explore Windows PowerShell examples from other sources, including the Internet, you will quickly realize that there are often many ways to accomplish a given task.

For example, in the previous example, you saw how to use New-Object and Add-Member to construct a custom object. Another approach would be to use Select-Object to add custom properties to a blank object:

```
function Get-Inventory {
    PROCESS {
        $computer = $_
        $os = gwmi win32_operatingsystem -comp $computer
        $bios = gwmi win32_bios -comp $computer
        $obj = new-object psobject
        $obj | select @{Label='ComputerName';Expression={$computer}},
            @{Label='SPVersion';Expression={$os.servicepackmajorversion}},
            @{Label='BIOSSerial';Expression={$bios.serialnumber}},
            @{Label='BuildNo';Expression={$os.buildnumber}}
    }
    gc names.txt | get-inventory
}
```

Neither approach is right or wrong; they both work, and they both accomplish the same result. You may feel that one or the other is easier to read or remember, and that is the approach you should use.

As you grow more comfortable with Windows PowerShell, you start to realize that many tasks can be accomplished with a single command, although that command may be very complex. For example, the entire Get-Inventory function shown previously could be replaced with a single complex command:

```
gwmi win32_operatingsystem -computer (gc names.txt) |
    select @{Label='ComputerName';Expression={$_.__SERVER}},
        @{Label='BuildNo';Expression={$_._BuildNumber}},
```

```
@{Label='SPVersion';Expression={$_.ServicePackMajorVersion}},  
@{Label='BIOSSerial';Expression={  
    (gwmi win32_bios -comp $.__SERVER).serialnumber  
}}
```

Here, the Get-WmiObject cmdlet is run first. Its –computerName parameter is being given the output of Get-Content, which is reading a text file that contains one computer name per line. The WMI object is piped to Select-Object.

The first three elements in Select-Object are creating custom object properties that use the properties from the Win32\_OperatingSystem WMI object. The idea is to provide custom property names, such as ComputerName and BuildNo, rather than using the native property names of the WMI class.

The last element in Select-Object is creating a custom property named BIOSSerial. That property's expression is actually executing a second Get-WmiObject cmdlet to retrieve the Win32\_BIOS class. The computer name being given to this

Get-WmiObject is the \_\_SERVER property from the first WMI object—that property contains the computer name. Notice that the entire Get-WmiObject command is in parentheses: This forces the shell to execute that command, and the parentheses represent the resulting object. The parentheses are followed by a period, indicating that we want to access one of the object's members, and then the member name SerialNumber. The end result is that the SerialNumber property of the Win32\_BIOS object is placed into the BIOSSerial custom property in Select-Object.

This is certainly a complex command, but it further demonstrates that many complex tasks can be accomplished in Windows PowerShell *without* formal scripting or programming. True, commands like this take some experience to read, let alone to construct from scratch. You are free to choose the approach that works best for you and that is easiest to remember.

## Advanced Functions

- An advanced function is a filtering function that specifies additional attributes for its input parameters
  - These attributes make it possible for the function to look and behave almost identically to a cmdlet written in a .NET Framework language
  - Advanced functions enables the creation of cmdlet-like structures that function similar to regular cmdlets

```
function Get-ComputerDetails {
    [CmdletBinding()]
    param (
        [parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [string[]]$computername
    )
    BEGIN {}
    PROCESS {}
    END {}
}
```

### Key Points

An advanced function is one step more advanced than a filtering function. Essentially, an advanced function is a filtering function that has additional attributes specified for its input parameters. These attributes make it possible for an advanced function to look and behave almost identically to a cmdlet written in a .NET Framework language.

Here is the function and parameter declaration for a very simple advanced function:

```
function Get-ComputerDetails {
    [CmdletBinding()]
    param (
        [parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [string[]]$computername
    )
    BEGIN {}
    PROCESS {
```

The remainder of this function will be demonstrated next. You will notice that the primary difference between this advanced function and the filtering functions you have already see are the extra attributes, such as [CmdletBinding()], the [parameter] attributes, and so on. In this example, the \$computername parameter has been designed to accept one or more strings as pipeline input and the parameter has been designated as mandatory.

Using this technique, it is not necessary to use the \$\_ placeholder within the function's PROCESS script block. Instead, any strings piped into this function are attached to the \$computername parameter. The PROCESS script block places one input object at a time into the \$computername parameter, and the PROCESS script block executes one time for each object that was piped in.

Advanced functions are the ultimate in Windows PowerShell modularization: Using cmdlet-style parameter attributes enables you to write a cmdlet-like structure that works very much like the shell's

regular cmdlets. You can even incorporate help capability into your functions, something you will learn about later in this course.

**Note:** Full coverage of advanced functions is beyond the scope of this course. However, a lesson providing more details on advanced functions is included in Appendix A, and you are encouraged to explore this information on your own.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Advanced Functions

- See an example of an advanced function



### Key Points

In this demonstration, you will see an example of an advanced function.

### Demonstration steps:

1. Start virtual machines **LON-DC1**, **LON-SVR1**, and **LON-SVR2** then log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Mod11\Democode\Advanced Functions.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lab E: Modularization

- Exercise 1: Generating an Inventory Audit Report

Logon information

Virtual machine	LON-DC1	LON-SVR1	LON-SVR2	LON-CLI1
Logon user name	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator	Contoso\Administrator
Password	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd	Pa\$\$w0rd

**Estimated time: 45 minutes**

### Estimated time: 45 minutes

You now have the knowledge and experience you need to begin crafting some very capable scripts to automate administrative activities. You will use those skills to create two scripts, one for generating an inventory report and another for testing network performance.

#### Lab Setup

Before you begin the lab you must:

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines. Wait until the boot process is complete.
2. Log on to the **LON-CLI1** virtual machine with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open the Windows PowerShell ISE. All PowerShell commands run within the Windows PowerShell ISE program.
4. Create a text file containing the names of all computers in the domain, one on each line, and save it as **C:\users\Administrator\documents\computers.txt**.

## Exercise 1: Generating an Inventory Audit Report

### Scenario

Your business is conducting an inventory audit and needs to generate a list of configuration information for several servers. This list must include the server's name, BIOS serial number, operating system version, service pack version, and total number of processor cores. This list must be in table format for management, in a CSV file for importation into a configuration database, and in an XML file for archival purposes.

The main tasks for this exercise are as follows:

1. Create a function called **Get-Inventory** that accepts computer names as pipeline input. This function must loop through each of the computers and use WMI to retrieve the BIOS serial number, operating system version, service pack version, and total number of processor cores. It returns a single object for each computer containing the server name and the properties retrieved from WMI.
2. Pass a list of all server names in your domain to the function and export the results of the function in three formats: .txt, .csv and .xml.

### ► Task 1: Create an inventory gathering function

1. Read the help documentation and examples for the **New-Object** cmdlet so that you understand how it can be used to generate new custom objects.
2. Read the help documentation in **about\_function** and the help options listed in here to learn more about functions and how they work.
3. Create a function called **Get-Inventory**. The **Get-Inventory** function should accept computer names from the pipeline. You do not need to declare a parameter for this.
4. In the body of the function create a **PROCESS** script block that iterates through the values piped into the function. Within that script block, retrieve BIOS, operating system and processor information using **Get-WmiObject**. It must then return a new object that contains the computer name, BIOS serial number, operating system version, service pack version and the number of processor cores.
5. Run this command to call your function:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory
```

**Hint:** You can enter that command as the last line in your script, underneath your function, and then run the script.

### ► Task 2: Export the results of the function in three formats: txt, csv and xml

1. Create a text file called **C:\inventory.txt** by piping the output of your **Get-Inventory** function to the **Out-File** cmdlet.
2. Create a csv file called **C:\inventory.csv** by piping the output of your **Get-Inventory** function to the **Export-Csv** cmdlet.
3. Create an xml file called **C:\inventory.xml** by piping the output of your **Get-Inventory** function to the **Export-Clixml** cmdlet.

**Results:** After this exercise, you should be able to create a pipeline function that processes information passed in from the pipeline and uses it to retrieve other information. You should also be able to combine multiple objects into one object and export data from a function in multiple formats.

## Lab Review

1. What parameter attributes are used to indicate that a parameter accepts pipeline input?
2. What is the advantage of using an advanced function instead of a regular function?

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

- What are some of the differences between the ForEach construct and the ForEach-Object cmdlet?
- Why do variable names begin with \$?
- What cmdlets are available to manage variables from other scopes?
- Can a parent scope ever read a variable from a child scope?
- What must you add to a cmdlet to ensure that any nonterminating errors it encounters will be trappable?
- What command can be used to output "trace," or debug messages? What must you include in a script to enable this output?
- What kinds of breakpoints can you set? Are breakpoints valid only in a script, or can they be used directly from the command line?
- What are the advantages of a filtering function over a parameterized function? What are the advantages of an advanced function over other types of functions?



### Review Questions

1. What some of the differences between the ForEach construct and the ForEach-Object cmdlet?
2. Why do variable names begin with \$?
3. What cmdlets are available to manage variables from other scopes?
4. Can a parent scope ever read a variable from a child scope?
5. What must you add to a cmdlet to ensure that any nonterminating errors it encounters are trappable?
6. What command can be used to output trace, or debug messages? What must you include in a script to enable this output?
7. What kinds of breakpoints can you set? Are breakpoints valid only in a script, or can they be used directly from the command line?
8. What are the advantages of a filtering function over a parameterized function? What are the advantages of an advanced function over other types of functions?

# Appendix A

## Additional Windows PowerShell® Features

### Contents:

<b>Lesson 1:</b> Additional Features Overview	A-3
<b>Lesson 2:</b> XML Features	A-5
<b>Lesson 3:</b> Advanced Functions	A-10
<b>Lesson 4:</b> Responding to Events	A-19
<b>Lesson 5:</b> Text Manipulation Features	A-23
<b>Lesson 6:</b> Using Windows Forms	A-26

MCT USE ONLY. STUDENT USE PROHIBITED

## Appendix Overview

- Explain the use of Windows PowerShell's XML features
- Create advanced functions that behave like cmdlets
- Write functions or scripts that respond to operating system events
- Describe and use Windows PowerShell's text-manipulation features, including regular expressions, the `-replace` operator, and `Select-String` cmdlet
- Use Windows Forms and Windows Presentation Framework to create a graphical dialog for use in a script



Windows PowerShell® includes dozens of additional features that are not covered in this course. Although all of these features are very useful in the right scenario, some are very advanced features that may only be of interest to developers, or to administrators who are more comfortable with system programming and scripting.

The purpose of this Appendix is to give you a starting point and examples for some of the additional shell features that might be of interest to an administrator. You can take these examples and starting points and discover more about these features on your own, by reading help topics within Windows PowerShell, by exploring additional examples on the Internet, and so forth.

Even these additional features do not, however, constitute the full set of shell capabilities. Some shell features—such as writing cmdlets in a .NET Framework language, or embedding the shell inside another application, or creating restricted runspaces—require advanced software development skills. Windows PowerShell offers a number of different features to a number of different audiences. If you find that you have a need for, or an interest in, other shell features, consider purchasing a book that covers those features in more detail.

### Appendix Objectives

After completing this module, you will be able to:

- Explain the use of Windows PowerShell's XML features.
- Create advanced functions that behave like cmdlets.
- Write functions or scripts that respond to operating system events.
- Describe and use Windows PowerShell's text-manipulation features, including regular expressions, the `-replace` operator, and `Select-String` cmdlet.
- Use Windows® Forms and Windows Presentation Framework to create a graphical dialog for use in a script.

## Lesson 1

# Additional Features Overview

- List and describe some of the additional Windows PowerShell features



Windows PowerShell contains numerous features beyond what was covered in the main portion of this course. Many of these features are suitable for more advanced administrators, or require more time to learn than was available during class. These are all excellent topics for you to explore and learn on your own, after you have mastered the fundamentals of using the shell for practical administrative tasks.

This Appendix will highlight some key features commonly of interest to administrators, provide examples of their use, and give you a starting point for learning more about them on your own.

There are no labs in this Appendix, but you should take the time to examine the demonstration scripts, and perhaps experiment with these features on your own.

### Lesson Objectives

After completing this lesson, you will be able to:

- List and describe some of the additional Windows PowerShell features.

MCT USE ONLY. STUDENT USE PROHIBITED

## Additional Features

- **XML Features.** PowerShell has the ability to read and interpret XML documents, and to present the document information in the form of an object hierarchy.
- **Advanced Functions.** You were briefly introduced to these in the main portion of the course. These functions are capable of behaving almost exactly like a cmdlet, although they are written using PowerShell's scripting language.
- **Events.** You were introduced to WMI events in the main portion of the course. You can also write PowerShell commands that are executed automatically in response to general operating system events.
- **Text Manipulation.** Although PowerShell is object-based, you may need to work with textual data from log files or other sources. The shell offers many features designed to parse and manipulate text.
- **Windows Forms.** Because it is based on .NET Framework, PowerShell can access the Windows Forms and WPF portions of .NET Framework, giving you the ability to create graphical elements for your shell scripts.

## Key Points

This Appendix will cover the following additional features in Windows PowerShell:

- **XML Features.** Windows PowerShell has the ability to read and interpret XML documents, and to present the document information in the form of an object hierarchy.
- **Advanced Functions.** You were briefly introduced to these in the main portion of the course. These functions are capable of behaving almost exactly like a cmdlet, although they are written using Windows PowerShell's scripting language.
- **Events.** You were introduced to WMI events in the main portion of the course. You can also write Windows PowerShell commands that are executed automatically in response to general operating system events.
- **Text Manipulation.** Although Windows PowerShell is object-based, you may need to work with textual data from log files or other sources. The shell offers many features designed to parse and manipulate text.
- **Windows Forms.** Because it is based on .NET Framework, Windows PowerShell can access the Windows Forms and Windows Presentation Framework portions of .NET Framework, giving you the ability to create graphical elements for your shell scripts.

## Lesson 2

# XML Features

- Describe and use Windows PowerShell's XML manipulation features



You have already learned to use Import-CliXML and Export-CliXML to work with XML data in the shell. Windows PowerShell also has the ability to import arbitrary, well-formed XML formats, and convert them into an object hierarchy that provides an easier way to access the data within the XML document.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe and use Windows PowerShell's XML manipulation features.

MCT USE ONLY. STUDENT USE PROHIBITED

## XML Documents

- The Extensible Markup Language (XML) is a generic set of rules that describe how to build text-based documents that can contain hierarchical data
- XML is not any one format
  - It is a grammar, a set of rules that describe how to build languages
- A use of XML is called an *XML application*
  - XHTML is an XML-based language used by Web browsers
  - Other XML applications exist for many other uses
- PowerShell can read almost any well-formed XML document, and convert that document to a generic object hierarchy for shell manipulation
  - It doesn't understand XML application rules, only XML rules

### Key Points

The Extensible Markup Language, or XML, is a generic set of rules that describe how to build text-based documents that can contain hierarchical data. XML itself is not any one format; rather, it is a *grammar*, or a set of rules that describe how to build languages. A particular use of XML is called an *XML application*. One example of an XML application is XHTML, which is an XML-based markup language understood by Web browsers to describe Web page layout and appearance. Other XML applications exist for thousands of other uses, such as storing data from retail transactions, storing math equations, delivering news feeds, and much more.

Windows PowerShell can read almost any well-formed XML document, and convert that document to a generic object hierarchy that you can manipulate within the shell. Windows PowerShell does not understand the specifics of any XML application; it simply understands the basic rules of XML, and uses those rules to read any document that complies with those rules.

## XML Hierarchy

- Loading an XML document into PowerShell uses a format similar to the one below, where content is cast to XML:

```
[xml]$xml = get-content example.xml
```

- Accessing data within the loaded hierarchy uses a bracketed numerical notation, such as:

```
$xml.computers.computer[0].name  
$xml.computers.computer[0].services.service[0].servicename
```

- Piping hierarchy objects to Get-Member will display child objects and properties:

```
$xml.computers | get-member
```

### Key Points

XML documents store data in a hierarchy, which helps to preserve the relationship between different pieces of data. Here is an example document:

```
<computers>  
  <computer name="localhost">  
    <services>  
      <service servicename="BITS" startmode="" startname="" />  
      <service servicename="W32Time" startmode="" startname="" />  
    </services>  
  </computer>  
  <computer name="server1">  
    <services>  
      <service servicename="BITS" startmode="" startname="" />  
    </services>  
  </computer>  
</computers>
```

If this document was stored in a file named example.xml, you could load it into the shell by using this command:

```
[xml]$xml = get-content example.xml
```

The [xml] type description forces the \$xml variable to be an XML document. The shell parses the contents of the file to create an object hierarchy from that document. The \$xml variable represents the document itself; individual nodes within the document are represented as object properties. To access the first computer's name, for example, you would run this command:

```
$xml.computers.computer[0].name
```

Or to access the name of the first service of the first computer:

```
$xml.computers.computer[0].services.service[0].servicename
```

You can pipe any object from this hierarchy to Get-Member to see child objects and properties:

```
$xml.computers | get-member
```

There are also methods for changing the data within the document, and for writing the changed document back to disk in a file.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with XML

- Learn how to use the XML features of Windows PowerShell



### Key Points

In this demonstration, you will learn how to use the XML features of Windows PowerShell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Appendix\Democode\Working with XML.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 3

# Advanced Functions

- Describe and create advanced functions in Windows PowerShell



You were briefly introduced to Advanced Functions in Module 9 of the course. An Advanced Function is a Windows PowerShell function that behaves like a cmdlet, offering named parameters, pipeline parameter binding, and other cmdlet-like features.

In this lesson, you will learn more specifics about Advanced Functions, especially setting up cmdlet-style parameters, supporting parameters such as `-whatif` and `-confirm`, and so on.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe and create advanced functions in Windows PowerShell.

## Functions Review

- Recall that the primary difference between a normal function and an advanced function are the cmdlet-style parameter definitions in advanced functions
  - This allows for named parameters, positional parameters, pipeline parameter binding, and simple data validation
- Advanced functions can support
  - -confirm
  - -whatif
  - -verbose
  - -ErrorAction
- PowerShell advanced functions provide great modularization without needing to resort to advanced software development skills or .NET Framework familiarity

### Key Points

The main difference between a normal function and an advanced function is that advanced functions use cmdlet-style parameter definitions. This allows for named parameters, positional parameters, and pipeline parameter binding, as well as simple parameter data validation.

Advanced functions can be defined as supporting the `-confirm` and `-whatif` parameters, enabling you to write functions that modify the system, while properly communicating their actions to the user in a way that's consistent with the rest of Windows PowerShell.

Advanced functions can also support common parameters, such as `-Verbose` and `-ErrorAction`.

In many ways, Advanced functions are the penultimate form of modularization in Windows PowerShell. They provide an experience consistent in almost every way with a cmdlet, but writing them does not require advanced software development skills or .NET Framework familiarity.

## Verbosity

- Advanced functions can respond to the –verbose switch without requiring special support

- Simply use Write-Verbose within your function to write verbose output:

```
Write-Verbose 'Beginning database query'
```

- If the function is run with the –verbose switch and the \$VerbosePreference variable is set to Continue, your verbose output will appear
- If the function is run without –verbose, your verbose output will be suppressed
  - You do not need to define the –verbose parameter; it exists automatically and is handled by the shell

### Key Points

Your advanced functions can respond to the –verbose switch without you having to do anything special to support it.

Simply use Write-Verbose within your function to write verbose output:

```
Write-Verbose 'Beginning database query'
```

If the function is run with –verbose, and the shell's \$VerbosePreference variable is set to Continue (rather than SilentlyContinue), then your verbose output will appear. If the function is run without –verbose, then your verbose output will be suppressed. You do not need to define the –verbose parameter; it exists automatically and is handled by the shell.

## Pipeline Parameter Binding

- Recall that the two types of pipeline parameter binding are **ByValue** and **ByPropertyName**
  - Advanced functions can support both binding types
  - Parameters are normally accessed within the function's PROCESS script block
  - Within that script block, your pipeline input parameter will contain only one value at a time
  - The PROCESS script block will execute once for each object that was piped into the function
- Refer to the Student Guide for two examples of functions that accept pipeline parameter binding **ByValue** and **ByPropertyName**

### Key Points

You have already learned about the two forms of pipeline parameter binding: **ByValue** and **ByPropertyName**. Advanced function parameters can support both types of binding.

Here is an example of a parameter that will use **ByValue** binding:

```
Function Win32Restart-Computer {  
    [CmdletBinding(  
        SupportsShouldProcess=$true,  
        ConfirmImpact="High"  
)  
    param (  
        [parameter(  
            Mandatory=$true,  
            ValueFromPipeline=$true  
)  
        [string[]]$computerName  
)  
    ...  
}
```

This defines a **-computerName** parameter that will accept one or more string values, and that will bind pipeline input by value. In other words, if string objects are piped into this function, those objects will be placed into the **-computerName** variable. You could run the function like this:

```
Get-Content names.txt | Win32Restart-Computer
```

The content from the **names.txt** file would be placed into the **-computerName** parameter.

**ByPropertyName** binding is also available. Here is the same function, with the **-computerName** parameter modified to also accept binding by property name:

```
Function Win32Restart-Computer {
```

```
[CmdletBinding(
    SupportsShouldProcess=$true,
    ConfirmImpact="High"
)]
param (
    [parameter(
        Mandatory=$true,
        ValueFromPipeline=$true,
        ValueFromPipelineByPropertyName=$true
    )]
    [string[]]$computerName
)
...
```

Now, the function will also look at incoming objects to see if they have a computerName property. If they do, that property's values will be attached to the -computerName parameter. For example:

```
Get-ADComputer -filter * |
    Select *,@{Label='ComputerName';Expression={$_.Name}} |
    Win32Restart-Computer
```

Here, the computer objects are given a ComputerName property that contains the same value as the native Name property. The objects are then piped to the advanced function.

When used with pipeline input, you will normally access your parameters within the function's PROCESS script block. Within that script block, your pipeline input parameter (-computerName in this example) will contain only one value at a time. The PROCESS script block will execute once for each object that was piped into the function.

## Supporting ShouldProcess

- Functions that modify the system in any way should be written to support the ShouldProcess shell capability
  - This capability engages when the `-whatif` and `-confirm` parameters are used
  - If your function is defined as supporting ShouldProcess, you do not need to declare `-whatif` or `-confirm`
- Declaring support for ShouldProcess is part of the function declaration:

```
Function Win32Restart-Computer {  
    [CmdletBinding(  
        SupportsShouldProcess=$true,  
        ConfirmImpact="High"  
)]
```

- Refer to the Student Guide for further details on the necessary syntax for supporting `-whatif` and `-confirm`

### Key Points

Whenever an advanced function will modify the system in some way, you should write the function to support the ShouldProcess capability of the shell. It is this capability that engages when the `-whatif` and `-confirm` parameters are used. If your function is defined as supporting ShouldProcess, then you do not need to declare the `-whatif` or `-confirm` parameters, because they are handled automatically. You do, however, need to make a special call before actually modifying the system, to ask the user if the function should proceed.

Declaring support for ShouldProcess takes place as part of the function declaration:

```
Function Win32Restart-Computer {  
    [CmdletBinding(  
        SupportsShouldProcess=$true,  
        ConfirmImpact="High"  
)]
```

In this declaration, you must specify a `ConfirmImpact` of *Low*, *Medium*, or *High*, which should roughly describe the potential severity of whatever your cmdlet is doing.

The shell-wide `$ConfirmPreference` variable determines whether or not the `-confirm` switch is automatically added when your function is run. If your `ConfirmImpact` is higher than the shell-wide `$ConfirmPreference`, then your function will automatically run as if `-confirm` was specified. In this example, if `$ConfirmPreference` were *Medium*, this function would always run as if `-confirm` has been specified.

Although the shell automatically defines `-confirm` and `-whatif`, you do need to take some special steps to actually implement them within your function. Locate whatever command or commands in your function actually implement the system change, and wrap those commands in a conditional statement:

```
if ($pscCmdlet.ShouldProcess($computername)) {  
    get-wmiobject win32_operatingsystem -computername $computername |
```

```
    invoke-wmimethod -name Win32Shutdown -argumentlist $_action  
}
```

The \$pscmdlet.ShouldProcess() function implements the –confirm and –whatif functionality. The function takes an argument, which you must provide. That argument should be the target of the system change—in this example, the name of a computer that will be restarted.

If the function was run with –whatif, then ShouldProcess() will always return \$False, preventing your action from occurring. The shell will automatically produce the *what if* output seen by the user.

If the function was run with –confirm, then ShouldProcess() will prompt the user for each target object, and will return \$True if the user indicates that the function should proceed with the action.

## Data Validation

- Additional parameter features can help the shell validate parameter values before the function runs:

```
[parameter(Mandatory=$true)]
[string]
[ValidateSet("Restart","LogOff","Shutdown","PowerOff")]
$action
```

- You can find more information and parameter options by reading the help files on advanced functions:

```
Help about_functions_advanced*
```

### Key Points

Additional parameter features can help the shell validate parameter values before the function actually runs. For example, you can use these features to ensure that only a specific set of values are accepted by the parameter:

```
[parameter(Mandatory=$true)]
[string]
[ValidateSet("Restart","LogOff","Shutdown","PowerOff")]
$action
```

This example also shows the syntax for defining a mandatory parameter.

You can find more information and parameter options by reading the built-in help files on advanced functions:

```
help about_functions_advanced*
```

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Advanced Functions

- Learn how to use various features of advanced functions



### Key Points

In this demonstration, you will learn how to use various features of advanced functions.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Appendix\Democode\Advanced Functions.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 4

# Responding to Events

- Describe the purpose of Windows operating system events
- Create a Windows Forms timer and enable it to send events
- Write Windows PowerShell commands that respond to events



In Module 4 you learned about WMI events, which are fired in response to certain activities that occur within WMI. You learned how to write Windows PowerShell commands that ran in response to those events, enabling you to have commands run in response to things like new processes being started.

Many other Windows components are capable of generating events, and in this lesson you will learn how to register to receive notification of them, and how to write commands that run in response to those notifications.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of Windows operating system events.
- Create a Windows Forms timer and enable it to send events.
- Write Windows PowerShell commands that respond to events.

MCT USE ONLY. STUDENT USE PROHIBITED

## Events

- An event is something that occurs within the operating system, often in response to some real-world interaction.
  - For example: Moving your mouse pointer over a control like a button or checkbox. Here, the OS generates *MouseMove* events.
  - Clicking a control generates a *Click* event.
  - Internal timers generate a *tick* event.
- PowerShell can register for notifications of certain events from certain objects.
  - You can then have the shell run commands when the event fires and the event notification is received.

## Key Points

An event is something that occurs within the operating system, often in response to some real-world interaction. For example, when you move your mouse pointer over a control like a button or a check box, the operating system generates *MouseMove* events to let the underlying control know that the mouse is over them. Clicking a control generates a *Click* event that a button or other control can respond to. Internal timers generate *tick* events that indicate a certain amount of time has elapsed.

Windows PowerShell can register for notifications of certain events from certain objects. You can then have the shell run commands when the event fires and the event notification are received.

## Creating a Timer

- The following Windows Forms timer is configured to *tick* every ten seconds
  - This script creates the timer and enables it, as timers are disabled by default:

```
# create a Windows Forms timer
$timer = new-object Timers.Timer

# set an interval to 10 seconds
$timer.interval = 10000
$timer.enabled = $true
```

- Nothing happens initially, because nothing has registered to receive those events or do anything with them

### Key Points

To demonstrate events, we will create a Windows Forms timer. We will set it to *tick* every ten seconds, and enable it (timers are disabled by default).

To do so:

```
# create a Windows Forms timer
$timer = new-object Timers.Timer

# set an interval to 10 seconds
$timer.interval = 10000
$timer.enabled = $true
```

Nothing happens initially. Although the operating system is generating events in the background, nothing has registered to receive those events or to do anything about them.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Working with Events

- Learn how to use events within the shell



### Key Points

In this demonstration, you will learn how to use events within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Appendix\Democode\Working with Events.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Lesson 5

# Text Manipulation Features

- Review methods of the String object
- Use the –replace, –split, and –join operators
- Use regular expression matching
- Use the Select-String cmdlet



Although Windows PowerShell is an object-oriented shell, Microsoft realized that administrators sometime have to work with textual data, such as log files. To support those activities, the shell includes a number of text-manipulation features. Some of these are fairly simple, while others can be very complicated and can support detailed text parsing and manipulation.

### Lesson Objectives

After completing this lesson, you will be able to:

- Review methods of the String object.
- Use the –replace, –split, and –join operators.
- Use regular expression matching.
- Use the Select-String cmdlet.

MCT USE ONLY. STUDENT USE PROHIBITED

## Text Manipulation

- The shell includes numerous text manipulation features, including:
  - Within the shell, strings are objects. Try piping *Hello* to Get-Member to see the various properties and methods that allow you to manipulate strings.
  - The –replace, –join, and –split operators provide easy ways to perform commonly-needed text manipulation functions.
  - Regular expressions use industry-standard regex syntax to perform pattern matching on strings.
  - The Select-String cmdlet can perform both simple and regular expression matches against text, including text that exists in files on disk.

### Key Points

The shell includes a number of features designed to support text manipulation, and you will see many of these used in a demonstration. The features include:

- Within the shell, strings are objects. Try piping *Hello* to Get-Member to see the various properties and methods that allow you to manipulate strings.
- The –replace, –join, and –split operators provide easy ways to perform commonly-needed text manipulation functions.
- Regular expressions use industry-standard regex syntax to perform pattern matching on strings.
- The Select-String cmdlet can perform both simple and regular expression matches against text, including text that exists in files on disk.



For more information on regular expression syntax, view the `about_regular_expressions` help topic within the shell. There are also third party sites available that can be very useful.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Manipulating Text

- Learn how to use various text manipulation techniques within the shell



### Key Points

In this demonstration, you will learn how to use various text manipulation techniques within the shell.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Appendix\Democode\Manipulating Text.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lesson 6

# Using Windows Forms

- Based on an example script, write a script that utilizes graphical user interface elements to collect and display user data



Sometimes you may have a need to write a Windows PowerShell script that displays and uses graphical elements, such as custom dialog boxes or other elements.

Administrators often used HTML Applications, or HTAs, in conjunction with Visual Basic Scripting Edition (VBScript) to accomplish this in the past. With Windows PowerShell, you have access to a much more robust graphical user interface (GUI) library: Windows Forms and Windows Presentation Framework (WPF).

### Lesson Objectives

After completing this lesson, you will be able to:

- Based on an example script, write a script that utilizes graphical user interface elements to collect and display user data.

## Windows Forms and WPF

- Microsoft .NET Framework contains a subset of elements called Windows Forms
  - These are used to construct GUI elements such as dialog boxes
- Newer versions of the .NET Framework include the Windows Presentation Framework (WPF)
  - WPF accomplishes the same thing as Windows Forms, but in a different manner
- A complete discussion on Windows Forms or WPF is outside the scope of this course
  - However, using examples in the following demo, you can see how Windows Forms work and can discuss WPF

### Key Points

Microsoft® .NET Framework contains a subset of elements called Windows Forms, which are used to construct GUI elements such as dialog boxes. Newer versions of the Framework include an additional subsystem called Windows Presentation Framework, or WPF, which accomplishes much the same thing, although in a different manner. With WPF, the definition for a GUI element's layout and appearance can be placed into a separate XML-formatted file, while the code that manipulates and implements the GUI can be in a Windows PowerShell script.

A complete discussion of either Windows Forms or WPF is outside the scope of this course. However, using examples, you can see how both work.

One difficulty in working with these technologies from within Windows PowerShell is that the shell does not provide a Visual Studio®-like visual form designer. Third parties, however, do provide tools that let you visually create a dialog box, and then produce the Windows PowerShell code necessary to implement that dialog box from within a script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Demonstration: Windows Forms

- Learn how to use GUI elements within a script



### Key Points

In this demonstration, you will see how to use GUI elements within a script.

### Demonstration steps:

1. Start and log on to virtual machine **LON-DC1** with username **Contoso\Administrator** and password **Pa\$\$w0rd**.
2. Refer to the demonstration script in virtual machine **LON-DC1** at **E:\Appendix\Democode\Windows Forms.ps1**.
3. Alternatively, the same file is also available as part of the companion content and you can refer to the steps and explanations there.

## Module 1: Fundamentals for Using Microsoft Windows PowerShell v2

# Lab A: Using Windows PowerShell As an Interactive Command-Line Shell

### Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. Open a Windows PowerShell session as **Administrator**. by doing the following:
  - a. **Go to Start > All Programs > Accessories > Windows PowerShell**.
  - b. Right click **Windows PowerShell** and click **Run as administrator**.
4. In the Windows PowerShell windows execute the following command:

**"These are my notes" | Out-File C:\users\Administrator  
\Documents\notes.txt**

### Exercise 1: Searching for Text Files

► **Task 1: Browse the local file system using familiar command prompt and/or UNIX commands**

1. In the PowerShell window, type the following command and press ENTER:

```
cd C:\
```

2. In the PowerShell window, type the following command and press ENTER:

```
dir C:\users
```

3. In the PowerShell window, type the following command and press ENTER:

```
cd users  
dir
```

4. In the PowerShell window, type the following command and press ENTER:

```
cd C:\
```

5. In the PowerShell window, type the following command and press ENTER:

```
chdir users  
ls
```

6. In the PowerShell window, type the following command and press ENTER:

```
cd \  
md test
```

7. In the PowerShell window type the following command and press ENTER:

```
rm test
```

► **Task 2: View the entire contents of a folder**

1. In the PowerShell window, type the following command and press ENTER:

```
dir C:\users\administrator -recurse
```

2. In the PowerShell window, type the following command and press ENTER:

```
dir C:\users\administrator -r
```

► **Task 3: View a list of all text files in all users' document folders**

1. In the PowerShell window, type the following command and press ENTER:

```
dir C:\users\*\documents -recurse -include *.txt
```

2. In the PowerShell window, type the following command and press ENTER:

```
notepad C:\users\administrator\documents\notes.txt
```

**Results:** After this exercise, you should have successfully navigated the file system, discovered the notes.txt file in the Administrator's user profile folder, and viewed the contents of that file in notepad.

## Exercise 2: Browsing the Registry

► **Task 1: View cmdlet help**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-PSDrive
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-PSDrive -Full | more
```

► **Task 2: Navigate the registry**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-PSDrive
```

2. In the PowerShell window ,type the following command and press ENTER:

```
cd HKLM:  
cd Software  
cd Microsoft
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-PSDrive -PSProvider Registry
```

4. In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
cd HKCU:  
dir
```

► Task 3: Create a new PSDrive

1. In the PowerShell window, type the following command and press ENTER:

```
New-PSDrive -Name HKU -PSProvider Registry -Root Registry::HKEY_USERS
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-PSDrive -PSProvider Registry
```

### Exercise 3: Discovering Additional Commands and Viewing Help

► Task 1: Discover commands using aliases and aliases using commands

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Alias dir
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Alias -Definition Get-ChildItem
```

► Task 2: Learn how to use the help system using Get-Help

1. In the PowerShell window, type the following command and press ENTER:

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

2. In the PowerShell window, type the following command and press ENTER:

```
help Get-Help -Examples
```

3. In the PowerShell window, type the following command and press ENTER:

```
help Get-Help -Full
```

► Task 3: Discover new commands with Get-Command

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Command *Service*
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-Command
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Command *Service* - CommandType cmdlet
```

4. In the PowerShell window, type the following command and press ENTER:

```
Get-Command -Noun Service
```

► **Task 4: Get current, online help for a command**

- In the PowerShell window, type the following command and press ENTER:

**Note:** If you receive a blocking message saying this site has been blocked by Internet Explorer Enhanced Security Configuration, you should add the site to your trusted sites in the Security tab of Internet Options or just click Add on the blocking dialogue that you received.

```
Get-Help Get-Service -Online
```

**Results:** After this exercise, you should know how to discover commands using aliases or wildcards, how to look up different parts of help information for a command, how to use help information to learn how to use a command, and how to get current help online.

## Exercise 4: Adding Additional Commands to Your Session

► **Task 1: Find all “Module” commands**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Command -Noun Module
```

► **Task 2: List all modules that are available**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

► **Task 3: Load the ServerManager module into the current session**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Import-Module
```

2. In the PowerShell window, type the following command and press ENTER:

```
Import-Module ServerManager
```

► **Task 4: View all commands included in the ServerManager module**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Command -Module ServerManager
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-WindowsFeature
```

**Results:** After this exercise, you should be able to view and load modules into PowerShell and show what commands they contain.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 5: Learning How to Format Output

### ► Task 1: View the default table format for a command

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Process w*
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Process w* | Format-Table
```

### ► Task 2: View the default list and wide formats for a command

1. In the PowerShell window, type the following command and press ENTER:

```
gps w* | Format-List
```

2. In the PowerShell window, type the following command and press ENTER:

```
gps w* | Format-Wide
```

**Results:** After this exercise, you should be able to show PowerShell output in table, list and wide formats using default views.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. What is the name of the cmdlet with the alias dir?  
**Answer:** Get-ChildItem.
2. How many Registry PSDrives are available in PowerShell by default?  
**Answer:** two; HKLM and HKCU.
3. How many commands are there with the noun Service?  
**Answer:** eight; Get-Service, New-Service, Restart-Service, Resume-Service, Set-Service, Start-Service, Stop-Service, Suspend-Service.
4. What command do you use to load modules into your current session?  
**Answer:** Import-Module (or ipmo if you use the alias).
5. How do you format PowerShell output in a list format?  
**Answer:** You pipe the output to Format-List.

# Lab B: Using the Windows PowerShell Pipeline

## Before You Begin

1. Start the **LON-DC1** virtual machine. You do not need to log on, but wait until the boot process is complete.
2. Start the **LON-CLI1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. Open a Windows PowerShell session as **Administrator**. by doing the following:
  - a. Go to **Start > All Programs > Accessories > Windows PowerShell**.
  - b. Right click **Windows PowerShell** and click on **Run as administrator**.

## Exercise 1: Stopping and Restarting a Windows Service

### ► Task 1: View the Windows Update service

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service "Windows Update"
```

### ► Task 2: Determine what would happen if you stopped the service

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service wuauserv | Stop-Service -whatif
```

### ► Task 3: Stop the Windows Update service with confirmation

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service wuauserv | Stop-Service -confirm
```

### ► Task 4: Restart the Windows Update service

- In the PowerShell window, type the following command and press ENTER:

```
gsv wuauserv | Start-Service
```

**Results:** After this exercise, you should have successfully stopped and restarted the Windows Update service and learned about how the `WhatIf` and `Confirm` common parameters can prevent accidental changes.

## Exercise 2: Exploring Objects Returned by PowerShell Commands

### ► Task 1: View cmdlet help

- In the PowerShell window, type the following command and press ENTER:

```
help Get-Member -Full
```

### ► Task 2: Get visible members for Service objects

- In the PowerShell window, type the following command and press ENTER:

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

► **Task 3: Get properties for Service objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Get-Member -MemberType Property
```

► **Task 4: Get all members for Service objects**

- In the PowerShell window, type the following command and press ENTER:

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

► **Task 5: Get base and extended or adapted members for Service objects**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Get-Member -View Base
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Get-Member -View Adapted,Extended
```

► **Task 6: Find properties by wildcard search and show them in a table**

1. In the PowerShell window, type the following command and press ENTER:

```
dir C:\ -Force
```

2. In the PowerShell window ,type the following command and press ENTER:

```
dir C:\ -Force | Get-Member *Time* -MemberType Property
```

3. In the PowerShell window, type the following command and press ENTER:

```
dir C:\ -Force | Format-Table Name,*Time
```

**Results:** After this exercise, you should be able to use Get-Member to discover properties and methods on data returned from PowerShell cmdlets and use that information to format tables with specific columns.

## Exercise 3: Processing PowerShell Output

► **Task 1: List all commands that are designed to process output**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Command -verb Out
```

► **Task 2: Show the default output for a command**

1. In the PowerShell window, type the following command and press ENTER:

```
help Get-EventLog -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog System -Newest 100 -EntryType Warning,Error | Out-Default
```

► **Task 3: Send command output to a file**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog System -Newest 100 -EntryType Warning,Error | Out-File  
C:\users\Administrator\Documents\RecentIssues.txt
```

► **Task 4: Send command output to a grid and sort and filter the results**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog System -Newest 100 -EntryType Warning,Error | Out-GridView
```

2. Click on the **InstanceId** column header twice; once to sort in ascending order, then a second time to sort in descending order.
3. Filter the results in the grid to show only one InstanceID.
  - a. Click on the **Add criteria** button.
  - b. Select the check box beside **InstanceId**.
  - c. Click on the **Add** button.
  - d. Click on the hyperlink **is less than or equal to operator** and change **operator** to **equals**.
  - e. Enter any event ID that you have in your result set in the <Empty> text box.
  - f. Click on the **X** in the top right corner of the grid view window to close it.

**Results:** After this exercise, you should know how to use PowerShell commands to redirect output to files or a grid view, and how to sort and filter data in the grid view.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. How can you use PowerShell to see what a command will do before you actually do it?

**Answer:** Use the -WhatIf common parameter.

2. What are the two most common categories of members on objects and what are they used for?

**Answer:** Properties and methods. Properties are used to store information about the object. Methods are used to perform an action using the object.

3. What command is used to show PowerShell data in a grid?

**Answer:** Out-GridView.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 2: Understanding and Using the Formatting System

# Lab: Using the Formatting Subsystem

### Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Start the **LON-SVR1** virtual machine.
3. Log on to **LON-SVR1 as Contoso\Administrator** with a password of **Pa\$\$w0rd**.
4. On the **Start** menu of LON-SVR1, click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**.

### Exercise 1: Displaying Calculated Properties

► **Task 1: Retrieve a list of processes**

- In the Windows PowerShell® window, type the following command and press ENTER:

```
Get-Process
```

► **Task 2: Display a table of objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Format-Table
```

► **Task 3: Select the properties to display**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Format-Table -Property CPU,Id,ProcessName
```

► **Task 4: Use calculated properties to show a custom property**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Format-Table -Property CPU,  
ID,ProcessName,@{Label="TotalMemory";Expression={$_.WS+$_.VM}}
```

**Results:** After this exercise, you will have displayed the CPU, ID and ProcessName values of all the running processes and have created a custom property adding the physical and virtual memory using calculated properties.

### Exercise 2: Displaying a Limited Number of Columns

► **Task 1: Retrieve a list of services**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service
```

► **Task 2: Display a specific number of columns from a collection of objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service|Format-Table -Property Status,Name
```

**Results:** After this exercise, you will have displayed a specific number of columns from all the installed services instead of displaying all their default properties.

## Exercise 3: Displaying All Properties and Values of Objects

► **Task 1: Retrieve a list of processes**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process
```

► **Task 2: Display every property and value for all the objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process|Format-List -Property *
```

**Results:** After this exercise, you will have displayed all of the properties of all of the running processes instead of displaying only their default properties.

## Exercise 4: Viewing Objects via HTML

► **Task 1: Retrieve a list of event log entries**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -LogName Application -Newest 25
```

► **Task 2: Convert a list of event log entries to HTML**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -LogName Application -Newest 25|ConvertTo-Html|Out-File -FilePath "C:\events.html"
```

► **Task 3: Use a custom title for an HTML page**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -LogName Application -Newest 25|ConvertTo-Html -Title "Last 25 events"|Out-File -FilePath "C:\events.html"
```

► **Task 4: View an HTML page in the default Web browser**

- In the PowerShell window, type the following command and press ENTER:

```
Invoke-Item -Path "C:\events.html"
```

**Results:** After this exercise, you will have displayed a customized HTML page that contains a listing of the 25 newest events from the Windows Application event log.

## Exercise 5: Displaying a Limited Number of Properties

► **Task 1: Display a list of files from a selected folder**

- In the PowerShell window, type the following command and press ENTER:

```
Get-ChildItem -Path "C:\Windows\System32" -Filter "au*.dll"
```

► **Task 2: Display a table with several different file properties**

- In the PowerShell window, type the following command and press ENTER:

```
Get-ChildItem -Path "C:\Windows\System32" -Filter "au*.dll" | Format-Table -Property Name,Length,Extension
```

**Results:** After this exercise, you will have displayed a table containing the Name, Length and Extension properties of all the files in C:\Windows\System32 that start with "au" and have the extension ".dll".

## Exercise 6: Displaying Objects Using Different Formatting

► **Task 1: Retrieve a list of services**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service
```

► **Task 2: Display a table showing only a few specific properties**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Format-Table -Property DisplayName,Status,DependantServices
```

► **Task 3: Display a table by eliminating any empty spaces between columns**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Format-Table -AutoSize -Property DisplayName,Status,DependantServices
```

**Results:** After this exercise, you will have displayed a table of all the installed services by displaying only the DisplayName, Status, and DependantServices properties, and you will have also removed any empty spaces between each column.

## Exercise 7: Displaying a Sorted List of Objects

► **Task 1: Retrieve a list of services**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service
```

► **Task 2: Display a table while sorting a specific property**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service|Sort-Object -Property Status|Format-Table
```

**Results:** After this exercise, you will have displayed a table of the installed services sorted by their Status property.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. It would be nice to be able to format the color of each row in some cases. Does Format-Table appear to provide the ability to change the color of rows?

**Answer:** No, Format-Table doesn't provide the ability to color rows within a table. Write-Host does provide coloring functionality though, but it can't be directly integrated with Format-Table.

2. What might happen if I used Select-Object in the pipeline, after I've used Format-Table (...|format-table <something>|select-object <something>)? Would that work?

**Answer:** No, Format-Table is meant to be used at the end of a pipeline. It actually modified the objects before displaying them, so using Select-Object on these modified objects won't work as expected.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 3: Core Windows PowerShell Cmdlets

# Lab A: Using the Core Cmdlets

### Before You Begin

1. Start the **LON-DC1**, **LON-SVR1**, and **LON-SVR2** virtual machines.
2. Log on to **LON-DC1**, **LON-SVR1**, and **LON-SVR2** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. Disable the Windows® Firewall for LON-DC1 by carrying out the following steps
  - a. Go to **Start > Control Panel > System and Security > Windows Firewall**.
  - b. Click **Turn Windows Firewall on or off**.
  - c. In the **Domain network location settings** section, click the radio button **Turn off Windows Firewall (not recommended)**.
  - d. In the **Home or work (private) network location** settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - e. In the **Public network location** settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - f. Click **OK**.
  - g. Repeat steps **a** to **e** for virtual machines LON-SVR1 and LON-SVR2.

**Note:** This is not security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on powershell concepts.

4. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

### Exercise 1: Sorting and Selecting Objects

► **Task 1: Retrieve a list of processes**

- On LON-SVR1, in the Windows PowerShell® window, type the following command and press ENTER:

`Get-Process`

► **Task 2: Sort a list of processes**

- In the PowerShell window, type the following command and press ENTER:

`Get-Process | Sort-Object`

► **Task 3: Select a certain number of processes**

- In the PowerShell window, type the following command and press ENTER:

`Get-Process | Sort-Object | Select-Object -First 5`

**Results:** After this exercise, you will have retrieved a list of first five services sorted alphabetically by their service name.

## Exercise 2: Retrieving a Number of Objects and Saving to a File

► **Task 1: Retrieve a list of event logs**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -List
```

► **Task 2: Retrieve a certain number of events**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -LogName Application -Newest 5
```

► **Task 3: Export a certain number of events to a file**

- In the PowerShell window, type the following command and press ENTER:

```
Get-EventLog -LogName Application -Newest 5 | Export-Csv -Path "C:\Export.csv"
```

**Results:** After this exercise, you will have exported the five most recent events from the Windows Application event log to a CSV file.

## Exercise 3: Comparing Objects Using XML

► **Task 1: Retrieve a list of services**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service
```

► **Task 2: Save a list of services to an XML formatted file**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Export-Clixml -Path "C:\Export.xml"
```

► **Task 3: Stop a service**

- In the PowerShell window, type the following command and press ENTER:

```
Stop-Service -Name "w32time"
```

► **Task 4: Compare services with different statuses**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Service | Export-Clixml -Path "C:\Export2.xml"
```

2. In the PowerShell window, type the following command and press ENTER:

```
Compare-Object -ReferenceObject (Import-Clixml -Path "C:\Export.xml") -DifferenceObject (Import-Clixml -Path "C:\Export2.xml") -Property Name,Status
```

► **Task 5: Restart a service**

- In the PowerShell window, type the following command and press ENTER:

```
Start-Service -Name "w32time"
```

**Results:** After this exercise, you will have determined that the Status property of two service comparison objects had changed after the Windows time service was modified. The difference between the files is the status of the W32Time service changing from running to stopped.

## Exercise 4: Saving Objects to a CSV file

► **Task 1: Retrieve a list of processes**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process
```

► **Task 2: Export a list of processes to a file**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Export-Csv -Path "C:\Export2.csv"
```

► **Task 3: Use a non-default separator**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Export-Csv -Path "C:\Export3.csv" -Delimiter ";"
```

**Results:** After this exercise, you will have exported a list of processes to a CSV file using the default separator ",", and the non-default separator ";".

## Exercise 5: Measuring a Collection of Objects

► **Task 1: Retrieve a list of processes**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process
```

► **Task 2: Measure the average, maximum and minimum of a collection**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process | Measure-Object -Property CPU -Average -Maximum -Minimum
```

**Results:** After this exercise, you will have measured the average, minimum and maximum values of the CPU property of all the running processes.

## Lab Review

1. Can you think of other delimiters you may want to use with Export-Csv?

**Answer:** Other examples could be "|" and ":".

2. Is there a difference between saving objects to a variable versus saving them to XML formatted file?

**Answer:** One thing XML formatted files are useful for, is being able to actually save the information of objects to disk, and even possibly transferring the information to another system.

3. In terms of objects having properties and methods, if an object is saved to a file, is anything lost?

**Answer:** One of the disadvantages of files versus variables is that even when the file is imported, the methods of the original object are basically lost.

# Lab B: Filtering and Enumerating Objects in the Pipeline

## Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Start the **LON-SVR1** virtual machine.
3. Start the **LON-SVR2** virtual machine.
4. Log on to **LON-SVR1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
5. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

## Exercise 1: Comparing Numbers (Integer Objects)

### ► Task 1: Compare using a simple expression

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
3 -gt 5
```

### ► Task 2: Compare using a more complex expression

- In the PowerShell window, type the following command and press ENTER:

```
(3 -lt 4) -and (5 -eq 5)
```

**Results:** After this exercise, you will have determined that 3 is smaller than 5 by having True returned, and also that the complex expression that compares whether 3 is smaller than 4 and 5 is equal to 5 resolves to True because both sides of the complex expression alone resolve to True.

## Exercise 2: Comparing String Objects

### ► Task 1: Compare using a simple expression

- In the PowerShell window, type the following command and press ENTER:

```
"PowerShell" -eq "powershell"
```

### ► Task 2: Compare using a more complex expression

- In the PowerShell window, type the following command and press ENTER:

```
("logfile" -eq "logfiles") -and ("host" -ceq "HOST")
```

**Results:** After this exercise, you will have determined that *PowerShell* is considered equal to *powershell* using the default comparison operators, and that the complex expression comparing *logfile* to *logfiles* and *host* to *HOST* resolves to False because one side of the complex expression alone resolves to False.

## Exercise 3: Retrieving Processes from a Computer

► **Task 1: Retrieve a list of processes**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process
```

► **Task 2: Retrieve objects from a remote computer**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Process -ComputerName "LON-DC1"
```

**Results:** After this exercise, you will have retrieved the list of processes running remotely on LON-DC1 from LON-SVR1.

## Exercise 4: Retrieving Services from a Computer

► **Task 1: Retrieve a list of services**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service
```

► **Task 2: Retrieve objects from a remote computer**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service -ComputerName "LON-DC1"
```

► **Task 3: Filter a collection of objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service -ComputerName "LON-DC1" -Name "w*"
```

**Results:** After this exercise, you will have retrieved a list of all the installed services that start with the letter "w" on LON-DC1 from LON-SVR1.

## Exercise 5: Iterating Through a List of Objects

► **Task 1: Create a file**

- Create a txt file named **C:\Systems.txt** that contains the following text: You can do this from the powershell command line if you are able or you can open Notepad and just enter the below data, each on a single line if you need.
  - **LON-SVR1.contoso.com**
  - **LON-SVR2.contoso.com**
  - **LON-DC1.contoso.com**
  - **LON-CLI1.contoso.com**

► **Task 2: Import the contents of a file**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Content -Path "C:\Systems.txt"
```

► **Task 3: Iterate through a list of objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Content -Path "C:\Systems.txt" | ForEach-Object -Process {$_.ToLower()}
```

► **Task 4: Filter a list of objects**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Content -Path "C:\Systems.txt" | Where-Object -FilterScript {$_.Name -like "*SVR*"}  
Results: After this exercise, you will have retrieved a list of entries from a file, have changed all the characters to lowercase, and also have filtered the list of entries to return only the ones with the string SVR in the entry name.
```

## Lab Review

1. What if you wanted to filter a list of processes that started with s\* and with e\*? Would you use Where-Object or –Name?

**Answer:** You would use Where-Object like this Get-Process|Where-Object{\$\_.Name –like "s\*" –or \$\_.Name –like "e\*"}  
–Name doesn't support regular expressions for a value.

2. When working with remote services, are there any advantages in using the –Name parameter versus using Where-Object?

**Answer:** Yes, the advantage of using –Name is that it has an effect similar to filtering. With something like Get-Service –Comp LON-DC1 –Name "a\*", only the services that start with a are retrieved. If this was done instead Get-Service –Comp LON-DC1|Where-Object{\$\_.Name –like "a\*"}, in this case, all of the services from the remote machine are retrieved and are processed locally as they are passed to Where-Object.

3. What might interfere with doing any kind of query of a remote computer?

**Answer:** Network firewalls, and even client side firewalls can interfere with remote queries like Get-Service and Get-Process.

# Lab C: Using Pipeline Parameter Bindingsvr

## Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Start the **LON-SVR1** virtual machine. You do not need to logon.
3. Start the **LON-SVR2** virtual machine. You do not need to logon.
4. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
5. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

## Exercise 1: Using Advanced Pipeline Features

### ► Task 1: Import the Active Directory module

- On LON-DC1, in the PowerShell window, type the following command and press ENTER:

```
Import-Module -Name ActiveDirectory
```

### ► Task 2: Import a CSV file

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "E:\Mod03\Labfiles\Module3_LabC_Exercise1.csv"
```

### ► Task 3: Create users in Active Directory

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "E:\Mod03\Labfiles\Module3_LabC_Exercise1.csv" | New-ADUser
```

**Results:** After this exercise, you will have automated the creation of 50 users in the contoso.com domain.

## Exercise 2: Working with Multiple Computers

### ► Task 1: Create a CSV file

- On LON-DC1, at the command prompt of the Windows PowerShell window, create a CSV file named "**C:\Inventory.csv**" that contains the following text:
  - **Type,Name**
  - **AD,LON-DC1**
  - **Server,LON-SVR1**
  - **Server,LON-SVR2**
  - **Client,LON-CLI1**

### ► Task 2: Import a CSV file and filter the results

- In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
Import-Csv -Path "C:\Inventory.csv" | Where-Object -FilterScript {$_._Type -eq "Server"}
```

► Task 3: Restart a group of computers

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "C:\Inventory.csv" | Where-Object -FilterScript {$_._Type -eq "Server"} | ForEach-Object -Process {Restart-Computer -ComputerName $_._Name}
```

**Note:** If someone is logged in to LON-SVR1 and LON-SVR2 you may receive an error message. To overcome the error message successful re-start the virtual machines, log off from them and then re-run the script.

**Results:** After this exercise, you will have restarted a filtered group of servers.

### Exercise 3: Stopping a List of Processes

► Task 1: Start two applications

- On LON-DC1, start the **Notepad.exe** and **Wordpad.exe** applications.

► Task 2: Create a CSV file

- At the command prompt of the Windows PowerShell window, create a CSV file named **C:\Process.csv** that contains the following text:

- Computer,ProcName
- LON-DC1,Notepad
- LON-DC1,Wordpad

► Task 3: Import a CSV file and iterate through a collection of objects

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "C:\Process.csv" | Select-Object -Property "ProcName"
```

► Task 4: Stop a list of processes

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "C:\Process.csv" | Select-Object -Property "ProcName" | ForEach-Object -Process {Stop-Process -Name $_._ProcName}
```

**Results:** After this exercise, you will have stopped a list of processes based on values imported from a CSV file.

### Exercise 4: Binding Properties to Parameters

► Task 1: Import a CSV file

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "E:\Mod03\Labfiles\Module3_LabC_Exercise4.csv"
```

### ► Task 2: Create users in Active Directory

- In the PowerShell window, type the following command and press ENTER:

```
Import-Csv -Path "E:\Mod03\Labfiles\Module3_LabC_Exercise4.csv" | Select-Object -  
Property @{Name="Name";Expression={$_.EmployeeName}},@{  
Name="SamAccountName";Expression={$_.EmployeeName}},@{Name="  
City";Expression={$_.Town}},@{ Name="Department";Expression={$_.Unit}},@{Name="  
EmployeeNumber";Expression={$_.Id }}|New-ADUser -Company "Contoso"
```

**Results:** After this exercise, you will have automated the creation of 20 users in the contoso.com domain.

## Lab Review

1. Do all cmdlets accept pipeline input like New-ADUser?

**Answer:** No, not all cmdlet parameters have been programmed this way. To check whether a parameter accepts pipeline input can be determined by checking Get-Help with the –Full switch.

2. What is an advantage when a cmdlet parameter accepts pipeline input?

**Answer:** A definite advantage is the shorter amount of typing required when the input can be read from any kind of input.

3. Before stopping a process, could a check be made to only stop the process if it was using more than a certain amount of physical or virtual memory?

**Answer:** Yes, simple using the filter cmdlet Where-Object in the pipeline could easily verify the current memory status. That would be useful in a script that is running regularly to check whether a process is having memory problems, and terminate it once it goes over a certain value.

## Module 4: Windows Management Instrumentation

# Lab: Using Windows Management Instrumentation in Windows PowerShell

### Before You Begin

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines.
2. Ensure that the Windows® Firewall on LON-DC1, LON-SVR1, LON-SVR2, and LON-CLI1 machines has been disabled by completing the following steps:
  - a. On LON -DC1 go to **Start > Control Panel > System and Security > Windows Firewall**.
  - b. Click on **Turn Windows Firewall on or off**.
  - c. In the **Domain network location settings** section, click the radio button **Turn off Windows Firewall (not recommended)**.
  - d. In the **Home or work (private) network location** settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - e. In the **Public network location** settings, click the radio button **Turn off Windows Firewall(not recommended)**.
  - f. Remain logged on to the virtual machine.
  - g. Repeat steps **a** to **e** for virtual machines LON-SVR1, LON-SVR2 and LON-CLI1.

**Note:** This is not security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on powershell concepts.

3. Log on to both **LON-CLI1** and **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
4. On the **Start** menu of LON-CLI1 and LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell** again.
5. On virtual machine LON-CLI1, create a text file containing the names of all computers in the domain, one on each line, and save it as **C:\users\administrator\documents\computers.txt**. You can either do this in the PowerShell command line if you are able or manually. Its contents should look as follows:

LON-DC1.contoso.com  
LON-SVR1.contoso.com  
LON-SVR2.contoso.com  
LON-CLI1.contoso.com

### Exercise 1: Building Computer Inventory

#### ► Task 1: Retrieve the operating system information

1. On LON-CLI1, in the Windows PowerShell® window, type the following command and press ENTER:

```
Get-WmiObject -Class Win32_OperatingSystem
```

2. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem | Format-List *
```

► **Task 2: Extract version information from the operating system object**

1. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem | fl *version*
```

2. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem | Format-Table  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion
```

**Note:** There are two underscores as a prefix to class SERVER.

► **Task 3: Retrieve the local operating system information “remotely”**

1. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName LON-CLI1 | Format-Table  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion
```

2. In the PowerShell window, type the following command and press ENTER:

```
get-help gwmi -parameter ComputerName
```

► **Task 4: Retrieve the operating system information for all computers**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Content C:\users\administrator\documents\computers.txt
```

2. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName (gc  
C:\users\Administrator\documents\computers.txt) | Format-Table  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion
```

► **Task 5: Export the operating system information for all computers**

1. In the PowerShell window, type the following command and press ENTER:

```
get-help select-object -full
```

2. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName (gc  
C:\users\Administrator\documents\computers.txt) | Select-Object  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion
```

3. In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName (gc  
C:\users\Administrator\documents\computers.txt) | Select-Object  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion | Export-Csv  
C:\Inventory.csv
```

► **Task 6: Retrieve the BIOS information for all computers**

- In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_BIOS -ComputerName (gc C:\users\Administrator\documents\computers.txt)
```

► **Task 7: Create a custom object for each computer containing all inventory information**

- In the PowerShell window, type the following command and press ENTER:

```
get-help select-object -full
```

- In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName (gc  
C:\users\Administrator\documents\computers.txt) | Select-Object  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion,@{name='AssetTag';ex  
pression={(gwmi Win32_BIOS -ComputerName $_.__SERVER).SerialNumber}}
```

► **Task 8: Export the inventory report**

- In the PowerShell window, type the following command and press ENTER:

```
gwmi Win32_OperatingSystem -ComputerName (gc  
C:\users\Administrator\documents\computers.txt) | Select-Object  
__SERVER,Version,ServicePackMajorVersion,ServicePackMinorVersion,@{name='AssetTag';ex  
pression={(gwmi Win32_BIOS -ComputerName $_.__SERVER).SerialNumber}} | Export-Csv  
C:\Inventory.csv
```

**Results:** After this exercise, you should have successfully retrieved operating system and BIOS information from all computers in your domain, built custom objects using Select-Object, and generated an inventory report in a .csv file.

## Exercise 2: Discovering WMI Classes and Namespaces

► **Task 1: View WMI classes in the default namespace**

- ON LON-DC1, in the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-WmiObject -Parameter List
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -List -ComputerName LON-DC1
```

► **Task 2: Find WMI classes using wildcards**

- In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Class *Printer* -List -ComputerName LON-DC1
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Namespace Root -Class *Printer* -List -Recurse -ComputerName LON-DC1
```

► **Task 3: Enumerate top-level WMI namespaces**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Namespace Root -Class __NAMESPACE -ComputerName LON-DC1
```

**Note:** There are two underscores as a prefix to class NAMESPACE.

2. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Namespace Root -Class __NAMESPACE -ComputerName LON-DC1 | Select-Object -ExpandProperty Name
```

**Results:** After this exercise, you should be able to find WMI classes in a specific namespace, find WMI classes using a wildcard search, and find WMI namespaces on local or remote computers.

## Exercise 3: Generating a Logical Disk Report for all Computers

► **Task 1: Discover the WMI class used to retrieve logical disk information**

- On LON-CLI1, in the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -class *disk* -list
```

► **Task 2: Retrieve logical disk information from the local machine**

- In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject Win32_LogicalDisk
```

► **Task 3: Retrieve logical disk information for hard disks in all computers in your domain**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-WmiObject -Parameter Filter
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject Win32_LogicalDisk -Filter 'DriveType=3' -ComputerName (gc C:\Users\Administrator\documents\computers.txt)
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject Win32_LogicalDisk -Filter 'DriveType=3' -ComputerName (gc C:\Users\Administrator\documents\computers.txt) | Format-Table SERVER,DeviceID,FreeSpace
```

**Note:** You may need to change the path and filename of the computers.txt file to correspond with the path and filename used on your computer.

MCT USE ONLY. STUDENT USE PROHIBITED

► **Task 4: Add a calculated PercentFree value to your report**

- In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject Win32_LogicalDisk -Filter 'DriveType=3' -ComputerName (gc  
C:\Users\Administrator\documents\computers.txt) | Format-Table  
__SERVER,DeviceID,FreeSpace,@{Label='PercentFree';Expression={$_.FreeSpace /  
$_.Size};FormatString='{0:#0.00%}'}
```

**Note:** For more information on the FormatString element and the formatting commands it accepts, see <http://go.microsoft.com/fwlink/?LinkId=193576>.

► **Task 5: Discover WMI information related to the logical drives**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject Win32_LogicalDisk -Filter 'DriveType=3' -ComputerName (gc  
C:\Users\Administrator\documents\computers.txt) | Select-Object -First 1
```

2. In the PowerShell window, type the following command and press ENTER:

```
(Get-WmiObject Win32_LogicalDisk -Filter 'DriveType=3' -ComputerName (gc  
C:\Users\Administrator\documents\computers.txt) | Select-Object -First  
1).GetRelated() | Format-Table __CLASS,__RELPATH
```

**Results:** After this exercise, you should know how to check logical disk free-space information using WMI, how to add calculated properties to a report, and how to find related WMI information from your WMI data.

## Exercise 4: Listing Local Users and Groups

► **Task 1: Find the WMI classes used to retrieve local users and groups**

1. On LON-CLI1, in the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -List *user*
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -List *group*
```

► **Task 2: Generate a local users and groups report**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Class Win32_UserAccount  
Get-WmiObject -Class Win32_UserAccount | fl *
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Class Win32_Group  
Get-WmiObject -Class Win32_Group | fl *
```

3. In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
Get-Help ForEach-object -full
```

4. In the PowerShell window, type the following command and press ENTER:

```
gc C:\users\administrator\documents\computers.txt | ForEach-Object {  
Get-WmiObject -Class Win32_UserAccount -Computer $_;  
Get-WmiObject -Class Win32_Group -Computer $_  
} | Format-Table -Property __CLASS,Domain,Name,SID
```

**Results:** After this exercise, you should be able to retrieve local user and group account information from local and remote computers and generate a report combining this information into a single table.

## Lab Review

1. How do you list WMI classes in a specific namespace?

**Answer:** Call Get-WmiObject passing the namespace into the Namespace parameter and using the List switch parameter—for example, Get-WmiObject –Namespace root\default -List.

2. What is the WMI class to retrieve operating system information called?

**Answer:** Win32\_OperatingSystem.

3. How do you use a server-side filter when using Get-WmiObject?

**Answer:** Pass a WMI expression to filter the data to the remote computer you are working with by using the Filter parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 5: Automating Active Directory® Administration

# Lab A: Managing Users and Groups

### Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. Disable the Windows® Firewall for LON-DC1 by carrying out the following steps
  - a. Go to **Start > Control Panel > System and Security > Windows Firewall**.
  - b. Click on **Turn Windows Firewall on or off**.
  - c. In the **Domain network location settings** section, click the radio button **Turn off Windows Firewall (not recommended)**.
  - d. In the **Home or work (private) network location** settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - e. In the Public network location settings, click the radio button **Turn off Windows Firewall (not recommended)**.

**Note:** This is not security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on powershell concepts. You can close the Windows Firewall dialogue when finished.

4. On the **Start** menu of LON-DC1, click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**. All commands will be run within the Windows PowerShell console.
5. In Windows PowerShell®, execute the following command:

```
Set-ExecutionPolicy RemoteSigned
```

### Exercise 1: Retrieving a Filtered List of Users from Active Directory

The main tasks for this exercise are:

1. List all modules installed on the local system.
2. Import the Active Directory module.
3. List all commands in the Active Directory module.
4. Retrieve all users matching a specific city and title by using server-side filtering.

#### ► Task 1: List all modules installed on the local system

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-Module -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

► **Task 2: Import the Active Directory module and populate the Active Directory environment**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Import-Module -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Import-Module ActiveDirectory
```

3. In the PowerShell window, type the following command and press ENTER:

```
E:\Mod05\Labfiles\Lab_05_Setup.ps1
```

► **Task 3: List all commands in the Active Directory module**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Command -Module ActiveDirectory
```

► **Task 4: Retrieve all users matching a specific city and title by using server-side filtering**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADUser -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser
```

3. When you are prompted to enter a value for the **Filter** parameter, type the following and click **OK**:

```
!?
```

4. After reviewing the help documentation for the **Filter** parameter, type the following and press ENTER:

```
office -eq "Bellevue"
```

5. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -Filter 'office -eq "Bellevue"'
```

6. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -Filter '(office -eq "Bellevue") -and (title -eq "PowerShell Scripter")'
```

**Results:** After this exercise you should have successfully imported the Active Directory module into PowerShell and used it to retrieve a filtered list of users from Active Directory.

## Exercise 2: Resetting User Passwords and Address Information

► **Task 1: Retrieve a list of remote users**

- In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
Get-ADUser -Filter 'office -ne "Bellevue"'
```

► Task 2: Reset remote user passwords to a specific password

1. In the PowerShell window, type the following command and press ENTER after each line:

```
Get-Help Read-Host -Full  
Get-Help Set-ADAccountPassword -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -Filter 'office -ne "Bellevue"' | Set-ADAccountPassword -Reset -  
NewPassword (Read-Host -AsSecureString 'New password')
```

3. When prompted enter the password **Pa\$\$w0rd** and press ENTER:

```
Pa$$w0rd
```

► Task 3: Change remote user address information to your local Bellevue, Washington office

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADUser -Parameter Properties
```

2. In the PowerShell window type the following command and press ENTER:

```
Get-ADUser -Filter 'office -ne "Bellevue"' -Properties  
Office,StreetAddress,City,State,Country,PostalCode | Format-Table  
SamAccountName,Office,StreetAddress,City,State,Country,PostalCode
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Set-ADUser -Full
```

4. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -Filter 'office -ne "Bellevue"' -Properties  
Office,StreetAddress,City,State,Country,PostalCode | Set-ADUser -Office Bellevue -  
StreetAddress '2345 Main St.' -City Bellevue -State WA -Country US -PostalCode  
'95102'
```

**Results:** After this exercise, you should be able to reset passwords and modify attributes for a filtered list of Active Directory users.

## Exercise 3: Disabling Users That Belong to a Specific Group

► Task 1: Retrieve a list of all Active Directory groups

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADGroup -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADGroup -Filter *
```

► **Task 2: Retrieve a specific Active Directory group named "CleanUp"**

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADGroup -Identity CleanUp
```

► **Task 3: Retrieve a list of members in the Active Directory group named "CleanUp"**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADGroupMember -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADGroup -Identity CleanUp | Get-ADGroupMember
```

► **Task 4: Disable the members of the Active Directory group named "CleanUp"**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Disable-ADAccount -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADGroup -Identity CleanUp | Get-ADGroupMember | Disable-ADAccount -WhatIf
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-ADGroup -Identity CleanUp | Get-ADGroupMember | Disable-ADAccount
```

**Results:** After this exercise, you should know how to retrieve groups from Active Directory, view their membership, and disable user accounts.

## Lab Review

1. Which common Active Directory cmdlet parameter is used to limit search results to matches based on attributes?

**Answer:** -filter.

2. Which common Active Directory cmdlet parameter is used to specify the attributes that you want in your query results?

**Answer:** Properties.

3. How do you add the Active Directory functionality to your PowerShell session?

**Answer:** ipmo ActiveDirectory

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab B: Managing Computers and Other Directory Objects

## Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Ensure that the Windows Firewall on LON-DC1 has been disabled.
3. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
4. On the **Start** menu of LON-DC1, click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**. All commands will be run within the Windows PowerShell console.

## Exercise 1: Listing All Computers That Appear to Be Running a Specific Operating System According to Active Directory Information

### ► Task 1: Import the Active Directory module

- In the PowerShell window, type the following command and press ENTER:

```
import-Module ActiveDirectory
```

### ► Task 2: List all of the properties for one AD computer object

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADComputer -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter * -Properties * -ResultSetSize 1 | fl *
```

### ► Task 3: Find any properties containing operating system information on an AD computer

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter * -Properties * -ResultSetSize 1 | fl *operatingsystem*
```

### ► Task 4: Retrieve all computers running a specific operating system

1. In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"'
```

**Note:** In some installations, the previous solution might not work, and might result in an error message regarding extended attributes. If that occurs, use the alternate solution provided next.

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties OperatingSystem
```

MCT USE ONLY. STUDENT USE PROHIBITED

**Results:** After this exercise you should be able to retrieve AD computers that match specific criteria and indicate which properties you want to retrieve for those computers.

## Exercise 2: Creating a Report Showing All Windows Server 2008 R2 Servers

► **Task 1: Retrieve all computers running Windows Server 2008 R2**

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties  
OperatingSystem,OperatingSystemHotfix,OperatingSystemServicePack,OperatingSystemVersi  
on
```

► **Task 2: Generate an HTML report showing only the computer name, SID, and operating system details**

- In the PowerShell window, type the following commands and press ENTER at the end of each line:

```
Get-Help ConvertTo-HTML -Full  
Get-Help Out-File -Full
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties  
OperatingSystem,OperatingSystemHotfix,OperatingSystemServicePack,OperatingSystemVersi  
on | ConvertTo-HTML -Property Name,SID,OperatingSystem* -Fragment
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties  
OperatingSystem,OperatingSystemHotfix,OperatingSystemServicePack,OperatingSystemVersi  
on | ConvertTo-HTML -Property Name,SID,OperatingSystem* | Out-File C:\OSList.htm
```

- In the PowerShell window, type the following command and press ENTER:

```
C:\OSList.htm
```

► **Task 3: Generate a CSV file showing only the computer name, SID, and operating system details**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Help Export-Csv -Full
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties  
OperatingSystem,OperatingSystemHotfix,OperatingSystemServicePack,OperatingSystemVersi  
on | Select-Object Name,SID,OperatingSystem*
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADComputer -Filter 'OperatingSystem -like "Windows Server 2008 R2*"' -Properties  
OperatingSystem,OperatingSystemHotfix,OperatingSystemServicePack,OperatingSystemVersi  
on | Select-Object Name,SID,OperatingSystem* | Export-Csv C:\OSList.csv
```

- In the PowerShell window, type the following command and press ENTER:

```
notepad C:\OSList.csv
```

**Results:** After this exercise, you should be able to generate HTML and CSV documents containing information that you retrieved by using PowerShell commands.

## Exercise 3: Discovering Any Organizational Units That Aren't Protected Against Accidental Deletion

► Task 1: Retrieve all organizational units

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-ADOrganizationalUnit -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADOrganizationalUnit -Filter * -Properties ProtectedFromAccidentalDeletion
```

► Task 2: Retrieve organizational units that are not protected against accidental deletion

- In the PowerShell window, type the following command and press ENTER:

```
Get-ADOrganizationalUnit -Filter * -Properties ProtectedFromAccidentalDeletion | Where-Object {-not $_.ProtectedFromAccidentalDeletion}
```

**Results:** After this exercise, you should be able to use the Active Directory cmdlets to retrieve organizational units from Active Directory.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. How can you see a list of all attributes that are available for an Active Directory object?  
**Answer:** Get-ADUser -Filter \* -Properties \* -ResultSetSize 1 | fl \*
2. Which parameter can be used to limit the total number of objects returned in an Active Directory query?  
**Answer:** ResultSetSize

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 6: Windows PowerShell Scripts

# Lab: Writing Windows PowerShell Scripts

### Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Start the **LON-SVR1** virtual machine.
3. Log on to **LON-SVR1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
4. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

### Exercise 1: Executing Scripts

#### ► Task 1: Check the execution policy setting

1. In the Windows PowerShell® window, type the following command and press ENTER:

```
Get-ExecutionPolicy
```

2. If the results from the previous command is "Restricted", in the PowerShell window, type the following command and press ENTER:

```
Set-ExecutionPolicy -ExecutionPolicy "RemoteSigned"
```

3. In the Execution Policy Change prompt type **Y** to indicate confirmation of your decision to change the policy:

#### ► Task 2: Create a script

- Start the Notepad application. Create a new file named **C:\Script.ps1** and enter the following text into the file. Save the file.

```
Get-Process -Name powershell
```

#### ► Task 3: Execute a script from the shell

- In the PowerShell window, type the following command and press ENTER after each line:

```
Set-Location -Path "C:\  
./Script.ps1
```

**Results:** After this exercise, you will have created and run a simple script that lists all the *powershell* processes.

### Exercise 2: Using Positional Script Parameters

#### ► Task 1: Copy commands into a script

- Start the Notepad application. Create a new file named **C:\Script2.ps1** and enter the following text into the file. Save the file.

```
Get-EventLog -LogName Application -Newest 100 | Group-Object -Property InstanceId | Sort-Object -Descending -Property Count | Select-Object -Property Name,Count -First 5 | Format-Table -AutoSize
```

► Task 2: Use commands with hardcoded values as parameter values

- In the PowerShell window, type the following command and press ENTER:

```
Set-Location -Path "C:\"
./Script2.ps1
```

► Task 3: Identify variable portions of a command

- The following underlined values are considered as variable names:

```
Get-EventLog -LogName Application -Newest 100 | Group-Object -Property InstanceId | Sort-Object -Descending -Property Count | Select-Object -Property Name,Count -First 5 | Format-Table -AutoSize
```

► Task 4: Create positional script parameters

- Start the Notepad application. Create a new file named **C:\Script3.ps1** and enter the following text into the file. Save the file.

```
Param($log,$new,$group,$first)
Get-EventLog -LogName $log -Newest $new | Group-Object -Property $group | Sort-Object -Descending -Property Count | Select-Object -Property Name,Count -First $first | Format-Table -AutoSize
```

► Task 5: Execute a script from the shell

- At the command prompt of the Windows PowerShell window, run the script just created and pass it four arguments.

```
Set-Location -Path "C:\"
./Script3.ps1 "Application" 100 "InstanceId" 5
```

**Results:** After this exercise, you will have created and run a script that uses positional parameters instead of using hardcoded values.

## Exercise 3: Using Named Script Parameters

► Task 1: Copy commands into a script

- Start the Notepad application. Create a new file named **C:\Script4.ps1** and enter the following text into the file. Save the file.

```
Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2 -MaxSamples 3
```

► Task 2: Use commands with hardcoded values as parameter values

- In the PowerShell window, type the following command and press ENTER:

```
Set-Location -Path "C:\"
./Script4.ps1
```

MCT USE ONLY. STUDENT USE PROHIBITED

► **Task 3: Identify variable portions of a command**

- The following underlined values are considered variable names:

```
Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2 -  
MaxSamples 3
```

► **Task 4: Create named script parameters**

- Start the Notepad application. Create a new file named **C:\Script5.ps1** and enter the following text into the file. Save the file.

```
Param($cpu,$interval,$max)  
Get-Counter -Counter "\Processor($cpu)\% Processor Time" -SampleInterval $interval -  
MaxSamples $max
```

► **Task 5: Execute a script from the shell**

- At the command prompt of the Windows PowerShell window, run the script just created by specifying all the parameter names and changing their order.

```
Set-Location -Path "C:\"  
. /Script5.ps1 -interval 2 -max 3 -cpu "_Total"
```

**Results:** After this exercise, you will have created and run a script that uses named parameters instead of using hard-coded values.

## Lab Review

1. If you send a script to another user with Windows 7 or Windows Server 2008 R2, will they be able to run the script? Is PowerShell included with these versions? Is it included with older Windows operating system versions?

**Answer:** Yes, both Windows 7 and Windows Server 2008 R2 come with PowerShell v2 by default. By default, their execution policy is set to Restricted, however, so a user won't be able to immediately run any scripts without changing the execution policy.

2. If you send a script to another user, how can the user incorporate the script so that it automatically runs every time PowerShell is opened?

**Answer:** By adding it to the profile, the user can make PowerShell load the script every time it is opened.

3. Is there a way to specify a default value for parameters in the param statement?

**Answer:** Yes, for example, by using the syntax `param([string]$my="value")`. In that case, unless a different value is provided when you invoke the script, \$my will be assigned the string "value".

## Module 7: Background Jobs and Remote Administration

# Lab A: Working with Background Jobs

### Before You Begin

1. Start the **LON-DC1** and **LON-SVR1** virtual machines.
2. Log on to **LON-SVR1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

### Exercise 1: Using Background Jobs with WMI

#### ► Task 1: Start a background job

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-Process -Name PowerShell}
```

#### ► Task 2: Check the status of a background job

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 1
```

**Note:** The job ID 1 might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

#### ► Task 3: Use Windows Management Instrumentation

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-WmiObject -Class Win32_ComputerSystem
```

#### ► Task 4: Retrieve information from remote computers

1. On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-WmiObject -ComputerName LON-DC1 -Class Win32_ComputerSystem}
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 3
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 3 | Receive-Job
```

**Results:** After this exercise, you will have used a background job to use WMI to retrieve the Win32\_ComputerSystem class from LON-DC1 remotely.

## Exercise 2: Using Background Jobs for Local Computers

► **Task 1: Start a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-Service}
```

► **Task 2: Check the status of a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 5
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Retrieve information from local commands**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 5 | Receive-Job
```

**Results:** After this exercise, you will have used a background job to retrieve a listing of all the installed services on LON-SVR1.

## Exercise 3: Receiving the Results from a Completed Job

► **Task 1: Run a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-EventLog -LogName Application -Newest 100}
```

► **Task 2: Check the status of a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 7
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Receive the results of a background job and keep the results in memory**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 7 | Receive-Job -Keep
```

- **Task 4: Remove the results of a background job from memory**
- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 7 | Receive-Job
```

**Results:** After this exercise, you will have used a background job to retrieve a listing of the latest 100 events in the Windows Application event log. You also will have seen how to save and remove the results from memory.

## Exercise 4: Removing a Completed Job

- **Task 1: Run a background job**
- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-HotFix}
```

- **Task 2: Check the status of a background job**
- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 9
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

- **Task 3: Remove a completed job**
1. On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Wait-Job -Id 9
```

2. On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Remove-Job -Id 9
```

**Results:** After this exercise, you will have run Get-HotFix as a background job, checked its status, and once completed, you will have removed the entire job from memory.

## Exercise 5: Waiting for a Background Job to Complete

- **Task 1: Start a background job**
- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Start-Sleep -Seconds 15}
```

- **Task 2: Check the status of a background job**
- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 11
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Wait for a background job to complete**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Wait-Job -Id 11
```

► **Task 4: Copy the results of a background job to a file**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Receive-Job -Id 11|Out-File -FilePath "C:\jobs.txt"
```

**Results:** After this exercise, you will have run Start-Sleep for 15 seconds, checked its status, and will have run the Wait-Job cmdlet to ensure the job was completed. You also will have copied the results of the background job to a local file.

## Exercise 6: Stopping a Background Job Before It Completes

► **Task 1: Start a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {while($true){Start-Sleep -Seconds 1}}
```

► **Task 2: Check the status of the background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 13
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Stop a background job that has not completed yet**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Stop-Job -Id 13
```

**Results:** After this exercise, you will have run a simple endless command as a background job, checked its status, and will have stopped the background job before it returned a completed status.

## Exercise 7: Working with the Properties of a Job

► **Task 1: Start a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Start-Job -ScriptBlock {Get-ChildItem -Path "C:\Windows\System32"}
```

► **Task 2: Display a list of background jobs**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job
```

► **Task 3: Use Get-Member to look at the properties of a job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 15 | Get-Member
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

**Results:** After this exercise, you will have started a background job that retrieves a listing of all the contents of C:\Windows\System32, used Get-Member to view the methods and properties of a background job, and will have displayed the ChildJobs property of the background job.

## Lab Review

1. You have started a PowerShell console and have started a couple of background jobs. What happens to the background jobs if they are still running and the main console is closed?

**Answer:** The background jobs will basically stop running completed and all the information they may contain will be lost.

2. If you have two or more PowerShell consoles open, and start a background job in one console, can you access the job from the other console?

**Answer:** No, jobs are tied directly to the console that started them and cannot be accessed by any other consoles.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab B: Using Windows PowerShell Remoting

## Before You Begin

1. Start the **LON-DC1**, **LON-SVR1**, and **LON-SVR2** virtual machines.
2. Log on to **LON-DC1**, **LON-SVR1**, and **LON-SVR2** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-DC1, LON-SVR1, and LON-SVR2 click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**.
4. In the Windows PowerShell window of LON-DC1, LON-SVR1, and LON-SVR2, run the command **Enable-PSRemoting -Force**.

## Exercise 1: Interactive Remoting

► **Task 1: Initiate a 1:1 remoting session to a remote computer**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Enter-PSSession -ComputerName LON-DC1
```

► **Task 2: Run remote commands interactively**

- In the PowerShell window, type the following command and press ENTER:

```
Get-HotFix
```

**Results:** After this exercise, you will have initiated a 1:1 remoting session with LON-DC1 from LON-SVR1 and will have issued Get-HotFix remotely.

## Exercise 2: Fan-Out Remoting

► **Task 1: Invoke a command on multiple computers**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

**Note:** If you are still in the remote session with LON-DC1 you can exit now by typing "Exit-psession".

```
Invoke-Command -ScriptBlock {Get-EventLog -LogName Application -Newest 100} - ComputerName LON-DC1,LON-SVR2
```

► **Task 2: Sort and filter objects in the pipeline**

- In the PowerShell window, type the following command and press ENTER:

```
Invoke-Command -ScriptBlock {Get-EventLog -LogName Application -Newest 100} - ComputerName LON-DC1,LON-SVR2|Group-Object -Property EventId|Sort-Object -Descending -Property Count|Select-Object -First 5
```

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 from LON-SVR1 to retrieve a listing of the 100 newest events from the Windows Application event log, and will have grouped and sorted the results to display the top five EventId values that have occurred the most.

## Exercise 3: Fan-Out Remoting Using Background Jobs

► **Task 1: Invoke a command on multiple computers as a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Invoke-Command -AsJob -ScriptBlock {Get-Acl -Path "C:\Windows\System32\*" -Filter "*.*.dll"} -ComputerName LON-DC1,LON-SVR2
```

► **Task 2: Wait for the parent job to complete**

- In the PowerShell window, type the following command and press ENTER:

```
Wait-Job -Id 1
```

**Note:** The job ID 1 might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Get the ID of all child jobs**

- In the PowerShell window, type the following command and press ENTER:

```
Get-Job -Id 1 | Select-Object -Property ChildJobs
```

► **Task 4: Sort and filter objects in the pipeline**

- In the PowerShell window, type the following command and press ENTER:

```
Receive-Job -Id 2,3 -Keep | Where-Object -FilterScript {$_.Owner -like "*Administrator*"}  
Get-Job -Id 1 | Select-Object -Property ChildJobs
```

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 as a background job from LON-SVR1 to retrieve the ACLs on all the files in C:\Windows\System32 with the extension ".dll", and will have filtered to display only the files with the owner being the Administrators.

## Exercise 4: Saving Information from Background Jobs

► **Task 1: Invoke a command on multiple computers as a background job**

- On LON-SVR1, in the PowerShell window, type the following command and press ENTER:

```
Invoke-Command -AsJob -ScriptBlock {Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2 -MaxSamples 3} -ComputerName LON-DC1,LON-SVR2
```

► **Task 2: Wait for the parent job to complete**

- In the PowerShell window, type the following command and press ENTER:

```
Wait-Job -Id 4
```

**Note:** The job ID number might be different on your computer. Run Get-Job with no parameters to see a list of jobs and their ID numbers.

► **Task 3: Receive the results only from a single child job**

- In the PowerShell window, type the following command and press ENTER:

```
Receive-Job -Id 5,6 -Keep | Where-Object -FilterScript {$_._PSComputerName -eq "LON-DC1"}
```

**Note:** The ID numbers of the items you want to use in the above command are the next two sequential numbers from the original job in task 1. If you wish to see these listed you can type or determine the ID numbers by typing `get-job -ID X | select-object -property childjobs` where X is the ID number from Task 1 above.

► **Task 4: Export the results to a file**

- In the PowerShell window, type the following command and press ENTER:

```
Receive-Job -Id 5,6 -Keep | Where-Object -FilterScript {$_._PSComputerName -eq "LON-DC1"} | Out-File -FilePath "C:\counter.txt"
```

**Results:** After this exercise, you will have invoked a command on LON-DC1 and LON-SVR2 as a background job from LON-SVR1 to retrieve the total %Processor Time value, waited for the parent job to complete, and will have retrieved the results of the child job that was used for LON-DC1 only by saving the results to a file.

## Lab Review

1. Is there any limit to the number of remote sessions that can be run?

**Answer:** Basically, the only limit is availability of resources on all the systems involved in PowerShell remoting. There is, however, a ThrottleLimit property available for Invoke-Command which limits the number of simultaneous child jobs that can run at any one time.

2. If I use Invoke-Command from one console and start it as a background job, can other consoles on the same server also see the background jobs and access their information directly?

**Answer:** No, all the background jobs are basically attached to the console that they were initiated from. You can, however, share information between consoles, but it requires using some intermediary, like saving the output from a job to a text or XML file, and importing that file into the other console.

## Module 8: Advanced Windows PowerShell Tips and Tricks

# Lab: Advanced PowerShell Tips and Tricks

### Before You Begin

1. Start the **LON-SVR1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, then right-click **Windows PowerShell** and click **Run As Administrator**.

### Exercise 1: Writing a Profile Script

#### ► Task 1: Write a profile script

1. In the Windows PowerShell® window, type the following command and press ENTER:

```
New-Item -Type File -Path $profile -Force
```

**Note:** You may need to run *Set-executionpolicy remotesigned* if that setting is not already defined.

2. In the PowerShell window, type the following command and press ENTER:

```
Notepad $profile
```

- If you get prompted to provide credentials you can use username **Contoso\Administrator** and password **Pa\$\$w0rd**.
3. In the Notepad window, type the following, and then click **Save** and **Exit**.

```
Set-Location -Path "C:\"
Import-Module -Name ServerManager
$cred=Get-Credential
```

#### ► Task 2: Test a profile script

- In the PowerShell window, type the following command and press ENTER:

```
. $profile
```

**Note:** As outlined earlier in Task 1 you may need to run *Set-executionpolicy remotesigned* if that setting is not already defined. Also, again If you get prompted to provide credentials you can use username **Contoso\Administrator** and password **Pa\$\$w0rd**.

**Hint:** That command is a period (or dot), a space, and then \$profile.

**Results:** After this exercise, you should have a better understanding on how to write and use a profile script.

## Exercise 2: Creating a Script Module

### ► Task 1: Package a function as a script module

1. Use Windows® Explorer to open your **Documents** (or **My Documents**) folder.
2. Double-click the **Windows PowerShell** folder. (If the folder does not exist, create it.)
3. Create a new subfolder named **Modules**. Double-click the new folder.
4. Create a new subfolder named **MyAppEvents**. Double-click the new folder.
5. Click **All Programs**, click **Accessories**, click **Windows PowerShell**, right-click **Windows PowerShell ISE** and click **Run As Administrator**
6. In the Windows PowerShell ISE open a new script file by pressing CTRL+N.
7. In the new script, type the following. Then save the file in the **MyAppEvents** folder that you created previously. Name the file **MyAppEvents.psm1**.

```
Function Get-AppEvents {  
    Get-EventLog -LogName Application -Newest 100 |  
        Group-Object -Property InstanceId |  
        Sort-Object -Descending -Property Count |  
        Select-Object -Property Name,Count -First 5 |  
        Format-Table -AutoSize  
}
```

### ► Task 2: Load a script module

- In the PowerShell window, type the following command and press ENTER:

**Note:** You may need to enter the full path to the file if you receive an error. You may also want to use the SPSHome directory location.

```
Import-Module -Name MyAppEvents
```

### ► Task 3: Test the script module

- In the PowerShell window, type the following command and press ENTER:

```
Get-AppEvents
```

**Results:** After this exercise, you should have a better understanding of how to create and import script modules.

## Exercise 3: Adding Help Information to a Function

### ► Task 1: Add comment based help to a function

1. In the Windows PowerShell ISE window, open the **MyAppEvent.psm1** file that you created in the previous exercise.
2. In the script, type the following, and then click **Save** and **Exit**. You may need to add only the boldfaced portion.

```
function Get-AppEvents {  
    <#  
    .SYNOPSIS  
    Get-AppEvents retrieves the newest 100 events from the Application  
    event log, groups and sorts them by InstanceID and then displays only  
    top 5 events that have occurred the most.  
    .DESCRIPTION  
    See the synopsis section.  
    .EXAMPLE  
    To run the function, simply invoke it:  
    PS> Get-AppEvents  
    #>  
    Get-EventLog -LogName Application -Newest 100|Group-Object -Property InstanceId|Sort-  
    Object -Descending -Property Count|Select-Object -Property Name,Count -First  
    5|Format-Table -AutoSize  
}
```

## ► Task 2: Test comment-based help

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-AppEvents
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-AppEvents -Examples
```

## ► Task 3: Reload a module

1. In the PowerShell window, type the following command and press ENTER:

```
Remove-Module -Name MyAppEvents
```

2. In the PowerShell window, type the following command and press ENTER:

```
Import-Module -Name MyAppEvents
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-AppEvents
```

**Results:** After this exercise, you should have a better understanding of how to add comment-based help to a function.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. What's the difference between adding comment-based help to a function versus adding lots of regular comments in the script itself?

**Answer:** Using comment-based help allows the Get-Help cmdlet to be used directly from the command line to get help information for functions that are included within modules. Otherwise, the user has to open the script file and read through nonstandardized help information, and that's only if the script author actually added comments.

2. Why is it important to refrain from using customized aliases in a module that is shared with others?

**Answer:** Because these customized aliases may not exist on other computers and, as a result, modules used by others may produce unexpected results.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 9: Automating Windows Server 2008 Administration

# Lab A: Using the Server Manager Cmdlets

### Before You Begin

1. Start the **LON-DC1**, **LON-SVR1** and **LON-SVR2** virtual machines.
2. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, right-click **Windows PowerShell** and click **Run As Administrator**.
4. On LON-DC1, open a Windows PowerShell® session as **Administrator**.
5. On LON-DC1, open the **Start** menu. Under the **Administrative Tools** folder, open the **Group Policy Management Console**.
6. In the left pane of the **Group Policy Management** console, expand the forest and domain until you see the **Starter GPOs** folder. Select the **Starter GPOs** folder by clicking on it.
7. In the right pane, click the **Create Starter GPOs Folder** button to create the Starter GPOs.
8. Ensure that Windows PowerShell remoting is enabled on LON-SVR1 and LON-SVR2. If you completed the previous labs in this course and have not shut down and discarded the changes in the virtual machines, remoting will be enabled. If not, open a Windows PowerShell session on those two virtual machines and run Enable-PSRemoting on each.
9. In Windows PowerShell, execute the following two commands:
  - a. **Set-ExecutionPolicy RemoteSigned**
  - b. **E:\Mod09\Labfiles\Lab\_09\_Setup.ps1**

**Note:** Some of the Group Policy Objects and DNS entries that this script will modify may already exist in the virtual machine and you may receive an error saying so if you have used the virtual machine during earlier modules and have not discarded those changes.

10. Log on to **LON-SVR1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
11. On the **Start** menu of LON-SVR1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, right-click **Windows PowerShell** and click **Run As Administrator**.
12. Disable the Windows® Firewall for LON-SVR1 by carrying out the following steps
  - a. Go to **Start > Control Panel > System and Security > Windows Firewall**.
  - b. Click on **Turn Windows Firewall on or off**.
  - c. In the **Domain network location settings** section, click the radio button **Turn off Windows Firewall (not recommended)**.
  - d. In the **Home or work (private) network** location settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - e. In the **Public network location** settings, click the radio button **Turn off Windows Firewall (not recommended)**.
  - f. Repeat steps a to e for virtual machines LON-DC1 and LON-SVR2.

**Note:** This is not security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on powershell concepts.

## Exercise 1: Listing All Currently Installed Features

► **Task 1: Import the Windows Server Manager PowerShell module**

1. On LON-SVR1, in the Windows PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Import-Module ServerManager
```

► **Task 2: View the module contents**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Command -module ServerManager
```

► **Task 3: View all installed server features**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WindowsFeature
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WindowsFeature | Where {$_.Installed}
```

## Exercise 2: Comparing Objects

► **Task 1: Import the XML file**

1. On LON-SVR1, in the Windows PowerShell window, type the following command and press ENTER:

```
$target=Import-Clixml -path E:\Mod09\Labfiles\TargetFeatures.xml
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
$current=Get-WindowsFeature
```

► **Task 2: Compare the imported features with the current features**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Compare-Object $target $current -property Installed,Name
```

## Exercise 3: Installing a New Server Feature

► **Task 1: View cmdlet help**

- On LON-SVR1, in the Windows PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
help Add-WindowsFeature -full
```

► **Task 2: Install missing features**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Compare-Object $target $current -property installed,name | where {$_.sideIndicator -eq ">" -and -not $_.installed} | foreach {Add-WindowsFeature $_.name}
```

**Note:** There are usually several ways to accomplish a task in Windows PowerShell. The solution here is but one possible choice. You may have a different solution that also works.

## Exercise 4: Exporting Current Configuration to XML

► **Task 1: Verify the current configuration**

- On LON-SVR1, in the Windows PowerShell window, type the following command and press ENTER:

```
Get-WindowsFeature | where {$_.installed}
```

► **Task 2: Export configuration to XML**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WindowsFeature | Export-Clixml $env:UserProfile\Documents\LON-SVR1-Features.xml
```

## Lab Review

1. What module do you need to import to manage server features?

**Answer:** ServerManager

2. What cmdlets are used to manage server features?

**Answer:** Get-WindowsFeature, Add-WindowsFeature and Remove-WindowsFeature

3. What type of object does Get-WindowsFeature write to the pipeline?

**Answer:** Microsoft.Windows.ServerManager.Commands.Feature

4. What are some of its properties?

**Answer:** Path, DisplayName, Name, Installed, Parent, and SubFeatures

5. Does the cmdlet support configuring a remote server?

**Answer:** No. Although you could use a PSSession to remotely manage features on another server.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab B: Using the Group Policy Cmdlets

## Before You Begin

1. Start the **LON-DC1** virtual machine.
2. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. Start the **LON-SVR1** virtual machine. You do not need to log on.
4. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

## Exercise 1: Listing All Group Policy Objects in the Domain

### ► Task 1: Import the Group Policy PowerShell module

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Import-Module GroupPolicy
```

### ► Task 2: View the module contents

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Command -module GroupPolicy
```

### ► Task 3: View Group Policy objects

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-GPO -all
```

## Exercise 2: Creating a Text-Based Report

### ► Task 1: Get selected Group Policy object properties

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Gpo -all | Sort ModificationTime | Select DisplayName,*time
```

### ► Task 2: Create a text file report

- In the Windows PowerShell window, type the following command and press ENTER:

```
get-gpo -all | Sort ModificationTime | Select DisplayName,*time | out-file \\LON-SVR1\E$\Mod09\Labfiles\Reports_GPOSummary.txt
```

## Exercise 3: Creating HTML Reports

### ► Task 1: Create a default Domain Policy report

- In the Windows PowerShell window, type the following one line command and press ENTER:

```
Get-GPOReport -Name "Default Domain Policy" -ReportType HTML -Path $env:UserProfile\Documents\DefaultDomainPolicy.htm
```

► Task 2: Create individual GPO reports

- In the Windows PowerShell window, type the following one line command and press ENTER:

```
Get-Gpo -all | foreach {Get-GPOReport -id $_.id -ReportType HTML -Path \\LON-SVR1\E$\Mod09\Labfiles\$($_.Displayname).htm}
```

## Exercise 4: Backing Up All Group Policy Objects

► Task 1: Back up the Default Domain policy

- In the Windows PowerShell window, type the following command and press ENTER:

```
Backup-Gpo -Name "Default Domain Policy" -Path $env:UserProfile\Documents
```

► Task 2: Back up all Group Policy objects

- In the Windows PowerShell window, type the following command and press ENTER:

```
Backup-GPO -All -Path \\LON-SVR1\E$\Mod09\Labfiles -Comment "Weekly Backup"
```

## Lab Review

1. What module do you need to import to manage Group Policy?

**Answer:** GroupPolicy

2. What cmdlet is used to get Group Policy information?

**Answer:** Get-GPO

3. What type of Group Policy reports can you create?

**Answer:** XML and HTML

4. Can you back up individual Group Policy objects?

**Answer:** Yes. Or you can back all Group Policy objects at the same time.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab C: Using the Troubleshooting Pack Cmdlets

## Before You Begin

1. Start the **LON-DC1** and **LON-SVR1** virtual machines. You do not need to log on.
2. Start the **LON-CLI1** virtual machine and log on as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-CLI1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, click **Windows PowerShell**.

## Exercise 1: Importing the Troubleshooting Pack Module

### ► Task 1: Import the Troubleshooting Pack PowerShell module

- On LON-CLI1, in the Windows PowerShell window, type the following command and press ENTER:

```
Import-Module TroubleShootingPack
```

### ► Task 2: View the module contents

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Command -module TroubleShootingPack
```

## Exercise 2: Solving an End-User Problem Interactively

### ► Task 1: Stop Windows search

- On LON-CLI1, in the Windows PowerShell window, type the following command and press ENTER:

```
Stop-Service WSearch
```

### ► Task 2: Run an interactive troubleshooting session

1. In the Windows PowerShell window, type the following command and press ENTER:

```
$pack=Get-TroubleshootingPack $env:windir\diagnostics\system\search
```

2. In the PowerShell window, type the following command and press ENTER:

```
Invoke-TroubleshootingPack $pack -Result "c:\result.xml"
```

3. Type **1** and Press ENTER.
4. Type **1** and press ENTER.
5. Type **x** and press ENTER. (it should be lowercase **x** that is used)

### ► Task 3: View the report

1. Click the **Internet Explorer** icon on the Taskbar.
2. In the Internet Explorer navigation field type **C:\result.xml** and press ENTER.

If prompted, answer yes to run scripts and allow blocked content.

3. Double-click files to review them. There will be several file sin the C:\Result.xml folder.
4. After reviewing the file, close Internet Explorer.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Solving a Problem Using Answer Files

### ► Task 1: Create an answer file

1. In the Windows PowerShell window, type the following one line command and press ENTER:

```
Get-Troubleshootingpack C:\Windows\diagnostics\system\search -AnswerFile  
SearchAnswer.xml
```

2. Press ENTER.
3. Type **1** and press ENTER.
4. Press ENTER.

### ► Task 2: Invoke troubleshooting using the answer file

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Stop-Service WSearch
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Invoke-TroubleshootingPack $pack -AnswerFile SearchAnswer.xml
```

3. Type **x** and press ENTER. (It should be lowercase **x** that is used.)

### ► Task 3: Invoke troubleshooting unattended

- In the Windows PowerShell window, type the following command and press ENTER:

```
Invoke-TroubleshootingPack $pack -AnswerFile SearchAnswer.xml -Unattended
```

### ► Task 4: Verify the solution

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Service WSearch
```

## Lab Review

1. What cmdlet do you use to load the Troubleshooting Pack?

**Answer:** Import-Module

2. What are the cmdlets in the Troubleshooting Pack?

**Answer:** Get-Troubleshootingpack, Invoke-Troubleshootingpack

3. Can you resolve a problem without using an answer file?

**Answer:** Yes.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab D: Using the Best Practices Analyzer Cmdlets

## Before You Begin

1. Start the **LON-DC1** virtual machine and log on as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
2. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**.

## Exercise 1: Importing the Best Practices Module

► **Task 1: Import the Best Practices PowerShell module**

1. On LON-DC1, in the Windows PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Import-Module BestPractices
```

► **Task 2: View the module contents**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Command -module BestPractices
```

## Exercise 2: Viewing Existing Models

► **Task 1: Display all available best practice models**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-BpaModel
```

## Exercise 3: Running a Best Practices Scan

► **Task 1: Run a best practices scan**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Invoke-BpaModel microsoft/windows/directoryservices
```

► **Task 2: View the scan results**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
$results=Get-BpaResult microsoft/windows/directoryservices
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
$results | more
```

3. Press the SPACEBAR to advance to the next page or type **Q** to quit.

► **Task 3: View selected results**

- In the Windows PowerShell window, type the following command and press ENTER:

```
$results | where {$_.severity -eq "Error"}
```

Depending on the current configuration, there may not be any errors to display.

► **Task 4: Create a best practices report**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
$results | sort Severity | Select  
ResultID,Severity,Category,Title,Problem,Impact,Resolution,Compliance | ConvertTo-  
Html -Title "Active Directory Best Practice Scan" -CssUri  
E:\Mod09\LabFiles\bpareport.css | Out-File $env:userprofile\documents\adbpa.htm
```

2. Click **Start**, click **All Programs** and then click **Internet Explorer**.
3. In the URL navigation field type **C:\Users\Administrator.NYC-DC1\Documents\adbpa.htm**.
4. Close Internet Explorer after viewing the file.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. What module do you need to import to run Best Practices scans?

**Answer:** BestPractices

2. What are some of the items that you can view in a scan result?

**Answer:** Severity, Category, Problem, Impact, Compliance and Help.

3. What are some of the Best Practice models available?

**Answer:**

Microsoft/Windows/DirectoryServices

Microsoft/Windows/DNSServer

Microsoft/Windows/WebServer

# Lab E: Using the IIS Cmdlets

## Before You Begin

1. Start the **LON-DC1** virtual machine and log on as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
2. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell**.

## Exercise 1: Importing the IIS Module

### ► Task 1: Import the IIS PowerShell module

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Module -ListAvailable
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Import-Module WebAdministration
```

### ► Task 2: View the module contents

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Command -module WebAdministration
```

## Exercise 2: Creating a New Web Site

### ► Task 1: Create a new Web site

1. In the Windows PowerShell window, type the following command and press ENTER:

```
mkdir $env:systemdrive\inetpub\PowerShellDemo
```

2. In the Windows PowerShell window, type the following one-line command and press ENTER:

```
Copy [path to lab folder]\UnderConstruction.html -destination  
$env:systemdrive\inetpub\PowerShellDemo\default.htm
```

3. In the Windows PowerShell window, type the following one-line command and press ENTER:

```
New-WebSite -Name PowerShellDemo -port 80 -PhysicalPath  
$env:systemdrive\inetpub\PowerShellDemo
```

### ► Task 2: View all Web sites

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Website
```

### ► Task 3: Verify the Web site

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Stop-Website "Default Web Site"
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Start-Website "PowerShellDemo"
```

3. On LON-DC1, click **Start**, click **All Programs** and then click **Internet Explorer**.
4. In the URL navigation field type:  
  
`http://lon-dc1`  
and press ENTER.
5. Close Internet Explorer after viewing the welcome page.

## Exercise 3: Backing Up IIS

► **Task 1: Back up IIS**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Backup-WebConfiguration Backup1
```

► **Task 2: Verify the backup**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WebConfigurationBackup Backup1
```

## Exercise 4: Modifying Web Site Bindings

► **Task 1: Display current binding information for all Web sites**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WebSite
```

**Note:** Because the Get-WebBinding cmdlet doesn't include the Web site name, you might use a command like this:

```
Get-WebSite | Foreach { $_ | add-member NoteProperty -name BindingInformation -value ((Get-WebBinding $_.Name).BindingInformation) -passthru } | Select Name,BindingInformation
```

► **Task 2: Change port binding**

- In the Windows PowerShell window, type the following one line command and press ENTER:

```
Get-WebSite | Foreach { $name=$_.Name ; Get-WebBinding $name | Set-WebBinding -Name $name -PropertyName Port -Value 8001 }
```

► **Task 3: Verify the new bindings**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WebSite
```

2. On LON-DC1, click **Start**, click **All Programs** and then click **Internet Explorer**.

3. In the URL navigation field type:

```
http://lon-dc1:8001
```

4. Close Internet Explorer.

## Exercise 5: Using the IIS PSDrive

► **Task 1: Connect to the IIS PSDrive**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-PSDrive
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
cd IIS:
```

► **Task 2: Navigate to the default site**

- In the Windows PowerShell window, type the following command and press ENTER:

```
cd "Sites\Default Web Site"
```

► **Task 3: Display all properties for the default site**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-Item . | select *
```

► **Task 4: Modify the default site**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-ItemProperty . -Name serverAutoStart -Value $False
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
cd C:
```

## Exercise 6: Restoring an IIS Configuration

► **Task 1: View existing backups**

- In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WebConfigurationBackup
```

► **Task 2: Restore the most recent configuration**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Restore-WebConfiguration -name Backup1
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-WebSite
```

**Note:** You can ignore any errors about denied access to  
c:\windows\system32\inetserv\config\scheman\wsmanconfig\_schema.xml.or  
c:\windows\system32\inetserv\mbschema.xml.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. What module do you need to import to manage IIS?

**Answer:** WebAdministration

2. What cmdlet do you use to create a new Web site?

**Answer:** New-WebSite

3. What cmdlet do you use to stop a Web site?

**Answer:** Stop-WebSite

4. What cmdlets do you use to modify Web site bindings?

**Answer:** Set-WebBinding

5. What do you see in the Sites folder of the IIS PSDrive?

**Answer:** A listing of all Web sites on the server such as Default Web Site

MCT USE ONLY. STUDENT USE PROHIBITED

## Module 11: Writing Your Own Windows PowerShell Scripts

# Lab A: Using Variables and Arrays

### Before You Begin

1. Start the **LON-DC1** and **LON-SVR1** virtual machines.
2. Log on to **LON-SVR1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-SVR1, click **All Programs**, click **Accessories**, click **Windows PowerShell**, and then click **Windows PowerShell ISE**.

### Exercise 1: Creating Variables and Interacting with Them

#### ► Task 1: Create and work with variables and variable types

1. In the Windows PowerShell® ISE window, type the following command and press ENTER:

```
$myValue = 1
```

2. In the PowerShell window, type the following command and press ENTER:

```
$myValue = 'Hello'
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Get-Variable -Full
```

4. In the PowerShell window, type the following command and press ENTER:

```
Get-Variable myValue | fl *
```

5. In the PowerShell window, type the following command and press ENTER:

```
Get-Variable myValue -ValueOnly
```

6. In the PowerShell window, type the following command and press ENTER:

```
[int]$myValue = 1
```

7. In the PowerShell window, type the following command and press ENTER:

```
$myValue = 'Hello'
```

#### ► Task 2: View all variables and their values

- In the PowerShell window, type the following command and press ENTER:

```
Get-Variable
```

#### ► Task 3: Remove a variable

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Help Remove-Variable -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Remove-Variable myValue
```

**Results:** After this exercise, you should be able to create new variables, view variable values, and remove variables.

## Exercise 2: Understanding Arrays and Hashtables

### ► Task 1: Create an array of services

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_arrays
```

2. In the PowerShell window, type the following command and press ENTER:

```
$services = Get-Service
```

3. In the PowerShell window, type the following command and press ENTER:

```
$services
```

4. In the PowerShell window, type the following command and press ENTER:

```
$services.GetType().FullName
```

### ► Task 2: Access individual items and groups of items in an array

1. In the PowerShell window, type the following command and press ENTER:

```
$services[0]
```

2. In the PowerShell window, type the following command and press ENTER:

```
$services[1]
```

3. In the PowerShell window, type the following command and press ENTER:

```
$services[-1]
```

4. In the PowerShell window, type the following command and press ENTER:

```
$services[0..9]
```

### ► Task 3: Retrieve the count of objects in an array

- In the PowerShell window, type the following command and press ENTER:

```
$services.Count
```

### ► Task 4: Create a hashtable of services

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_hash_tables
```

2. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable = @[]
```

3. In the PowerShell window, type the following command and press ENTER:

```
foreach ($service in $services) {  
    $serviceTable[$service.Name] = $service  
}
```

4. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable
```

► **Task 5: Access individual items in a hashtable**

1. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable['BITS']
```

2. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable['wuauserv']
```

3. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable | gm
```

4. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable.Count
```

► **Task 6: Enumerate key and value collections of a hashtable**

1. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable.Keys
```

2. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable.Values
```

► **Task 7: Remove a value from a hashtable**

1. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable.Remove('BITS')
```

2. In the PowerShell window, type the following command and press ENTER:

```
$serviceTable['BITS']
```

**Results:** After this exercise, you should be able to create arrays and hashtables, access individual items in arrays and hashtables, and remove items from hashtables.

## Exercise 3: Using Single-Quoted and Double-Quoted Strings and the Backtick

### ► Task 1: Learn the differences between single-quoted and double-quoted strings

1. In the PowerShell window, type the following command and press ENTER:

```
$myValue = 'Hello'
```

2. In the PowerShell window, type the following command and press ENTER:

```
"The value is $myValue"
```

3. In the PowerShell window, type the following command and press ENTER:

```
'The value is not $myValue'
```

### ► Task 2: Use a backtick (`) to escape characters in a string

1. In the PowerShell window, type the following command and press ENTER:

```
"Nor is the value `\$myValue"
```

2. In the PowerShell window, type the following command and press ENTER:

```
dir 'C:\Program Files'
```

3. In the PowerShell window, type the following command and press ENTER:

```
dir C:\Program` Files
```

### ► Task 3: Use a backtick (`) as a line continuance character in a command

- In the PowerShell window, type the following command and press ENTER:

```
Get-Service `n-name "Windows Update"
```

**Results:** After this exercise, you should be able to create simple and compound strings and use a backtick to split a single command across multiple lines.

## Exercise 4: Using Arrays and Array Lists

### ► Task 1: Learn how to extract items from an array

1. In the PowerShell window, type the following command and press ENTER:

```
$days = @('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')
```

2. In the PowerShell window, type the following command and press ENTER:

```
$days[2]
```

### ► Task 2: Discover the properties and methods for an array

1. In the PowerShell window, type the following command and press ENTER:

```
$days | Get-Member
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Member -InputObject $days
```

**Results:** After this exercise, you should be able to extract specific items from arrays by index or from array lists by name. You should also be able to discover the properties and methods for an array or array list.

## Exercise 5: Using Contains, Like and Equals Operators

► **Task 1: Use contains, like and equals operators to find elements in arrays**

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_comparison_operators
```

2. In the PowerShell window, type the following command and press ENTER:

```
$colors =  
@('Red', 'Orange', 'Yellow', 'Green', 'Black', 'Blue', 'Gray', 'Purple', 'Brown', 'Blue')
```

3. In the PowerShell window, type the following command and press ENTER:

```
$colors -like '*e'
```

4. In the PowerShell window, type the following command and press ENTER:

```
$colors -like 'b*'
```

5. In the PowerShell window, type the following command and press ENTER:

```
$colors -like '*a*'
```

6. In the PowerShell window, type the following command and press ENTER:

```
$colors -contains 'White'
```

7. In the PowerShell window, type the following command and press ENTER:

```
$colors -contains 'Green'
```

8. In the PowerShell window, type the following command and press ENTER:

```
$colors -eq 'Blue'
```

► **Task 2: Understand the difference when using those operators with strings**

1. In the PowerShell window, type the following command and press ENTER:

```
$favoriteColor = 'Green'
```

2. In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
$favoriteColor -like '*e*' 
```

3. In the PowerShell window, type the following command and press ENTER:

```
$favoriteColor -contains 'e' 
```

**Results:** After this exercise, you should be able use the contains, like, and equal operators to show partial contents of a collection. You should also understand how these operators function differently, depending on the type of object they are being used with.

## Lab Review

1. How do you retrieve the members (properties and methods) for a collection?

**Answer:** Use Get-Member with the InputObject parameter and pass the collection to InputObject.

2. When should you use a double-quoted string instead of a single-quoted string?

**Answer:** When you want the variables and/or subexpressions inside the string expanded and the resulting values placed in the string.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab B: Using Scripting Constructs

## Before You Begin

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines.
2. Log on to **LON-DC1** as **Contoso\Administrator** with a password of **Pa\$\$w0rd**.
3. On the **Start** menu of LON-DC1 click **All Programs**, click **Accessories**, click **Windows PowerShell**, and click **Windows PowerShell ISE**.
4. On LON-CLI1, disable Windows Firewall by carrying out the following steps:
  - a. Go to **Start > Control Panel > System and Security > Windows Firewall**.
  - b. Click **Turn Windows Firewall on or off**.
  - c. In the **Domain network location settings** section, click **Turn off Windows Firewall (not recommended)**.
  - d. In the **Home or work (private) network location** settings, click **Turn off Windows Firewall (not recommended)**.
  - e. In the **Public network location** settings, click **Turn off Windows Firewall (not recommended)**.
  - f. Repeat steps a through e for virtual machines LON-DC1, LON-SVR1, and LON-SVR2.

**Note:** Disabling the firewall is not a security best practice and under no circumstances should this be done in a production environment. This is done in this instance purely to allow us to focus on the learning task at hand and concentrate on PowerShell concepts.

## Exercise 1: Processing and Validating Input

### ► Task 1: Read input from the end user

1. In the PowerShell ISE window, type the following command and press ENTER:

```
Get-Help Read-Host -Full
```

2. In the PowerShell window, type the following command and press ENTER:

```
Read-Host -Prompt 'Enter anything you like and press <ENTER> when done'
```

You will receive the Windows PowerShell ISE – Input dialogue bearing the text you outlined above in Step 2. Enter any text you like and click **OK**.

3. In the PowerShell window, type the following command and press ENTER:

```
$serverName = Read-Host -Prompt 'Server Name'
```

Again you will receive the Windows PowerShell ISE – Input dialogue bearing the text you outlined above in Step 3. Enter any text you like and click **OK**.

### ► Task 2: Validate input received from the end user

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_if
```

2. In the PowerShell window, type the following command and press ENTER

```

if ($serverName -notlike 'SERVER*') {
"Server name '$serverName' is REJECTED"
} else {"Server name '$serverName' is OK!"
}

```

- Run the command if it has not already run by pressing Enter in the last step. You can run the commands by highlighting the code you wish to run, right clicking and going to "Run Selection" or by pressing F5. In the resultant diaogue type any text and click **OK**.

► **Task 3: Create a script that prompts for specific input and exits only when the right input is found or when no input is provided**

- In the PowerShell window, type the following command and press ENTER:

```
get-help about_do
```

- In the PowerShell window, type the following command and press ENTER:

```

do {
cls
$serverName = Read-Host -Prompt 'Server Name (this must start with "SERVER")'
} until ((-not $serverName) -or ($serverName -like 'SERVER*'))

```

- In the PowerShell window, type the following command and press ENTER:

```
$serverName
```

- In the PowerShell window, type the following command and press ENTER:

```

do {
cls
$serverName = Read-Host -Prompt 'Server Name (this must start with "SERVER")'
} until ((-not $serverName) -or ($serverName -like 'SERVER*'))
$serverName

```

- Again run the command if it has not already run by pressing Enter in the last step. You can run the commands by highlighting the code you wish to run, right clicking and going to "Run Selection" or by pressing F5. In the resultant diaogue type any text and click **OK**.

**Results:** After this exercise, you should be able to read and process user input in a PowerShell script.

## Exercise 2: Working with For, While, Foreach, and Switch

► **Task 1: Create a basic for loop**

- In the PowerShell window, type the following command and press ENTER:

```
$i = 100
```

- In the PowerShell window, type the following command and press ENTER:

```
Get-Help about_For
```

- In the PowerShell window, type the following command and press ENTER:

```
for ($i; $i -lt 100; $i++) {  
    Write-Host $i  
}
```

4. In the PowerShell window, type the following command and press ENTER:

```
for ($i = 0; $i -lt 100; $i++) {  
    Write-Host $i  
}
```

► **Task 2: Create a basic while loop**

1. In the PowerShell window, type the following command and press ENTER:

```
$i = 0
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help about_While
```

3. In the PowerShell window, type the following command and press ENTER:

```
while ($i -lt 100) {  
    write $i; $i++ }
```

► **Task 3: Use the for statement to create users in Active Directory**

1. In the PowerShell window, type the following command and press ENTER:

```
Import-Module ActiveDirectory
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help New-ADUser
```

3. In the PowerShell window, type the following command and press ENTER:

```
New-ADUser -Name "Module11BTest1" -Path 'CN=Users,DC=contoso,DC=com' -SamAccountName  
"Module11BTest1" -PassThru
```

4. In the PowerShell window, type the following command and press ENTER:

```
$numUsers = 10
```

5. In the PowerShell window, type the following command and press ENTER:

```
for ($i = 2; $i -le $numUsers; $i++) {  
    New-ADUser -Name "Module11BTest$i" -Path 'CN=Users,DC=contoso,DC=com' -SamAccountName  
    "Module11BTest$i" -PassThru  
}
```

6. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -filter 'name -like "Module11BTest*"' | Remove-ADUser
```

When you are prompted to confirm whether or not you want to remove the users, press **A** to remove all test users.

MCT USE ONLY. STUDENT USE PROHIBITED

► **Task 4: Use foreach to iterate through a collection of objects**

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_foreach
```

2. In the PowerShell window, type the following command and press ENTER:

```
$disks = Get-WmiObject Win32_LogicalDisk
```

3. In the PowerShell window, type the following command and press ENTER:

```
foreach ($disk in $disks) {  
    $disk | Select-Object DeviceID,DriveType  
}
```

► **Task 5: Use switch to test a value and translate it to a human-readable string**

1. In the PowerShell window, type the following command and press ENTER:

```
get-help about_switch
```

2. In the PowerShell window, type the following command and press ENTER:

```
foreach ($disk in $disks) {  
    $driveTypeValue = switch ($disk.DriveType) {  
        0 {  
            'Unknown'  
            break  
        }  
        1 {  
            'No Root Directory'  
            break  
        }  
        2 {  
            'Removable Disk'  
            break  
        }  
        3 {  
            'Local Disk'  
            break  
        }  
        4 {  
            'Network Drive'  
            break  
        }  
        5 {  
            'Compact Disk'  
            break  
        }  
        6 {  
            'RAM Disk'  
            break  
        }  
    }  
    $disk |  
    Select-Object -Property DeviceID,  
    @{name='DriveType';expression={$driveTypeValue}}  
}
```

**Note:** Be aware of the formatting if you receive errors. This will execute if typed exactly as above. You should enter this into the PowerShell ISE or you can type it all on one line and use semi-colon ; before the break command.

i.e .....6 {'RAM Disk'; break }.....

3. Run the commands if it has not already run by pressing Enter in the last step. You can run the commands by highlighting the code you wish to run, right clicking and going to "Run Selection" or by pressing F5. In the resultant diaogue type any text and click **OK**.

**Results:** After this exercise, you should be able to use the for, foreach, and switch statements in your PowerShell scripts.

### Exercise 3: Using the Power of the One-Liner

► **Task 1: Create scripts to retrieve and process data**

1. In the PowerShell window, type the following command and press ENTER:

```
foreach ($computer in @('LON-DC1','LON-SVR1','LON-SVR2','LON-CLI1')) {  
    Get-Service -Name wuauserv -ComputerName $computer  
}
```

2. In the PowerShell window, type the following command and press ENTER:

```
foreach ($user in Get-ADUser -Filter * -Properties Office) {  
    if ($user.Office -eq 'Bellevue') {  
        $user  
    }  
}
```

► **Task 2: Create one-liners for those scripts that are much more efficient**

1. In the PowerShell window, type the following command and press ENTER:

```
Get-Service -Name w* -ComputerName @('LON-DC1','LON-SVR1','LON-SVR2','LON-CLI1')
```

**Note:** Remember that the @ and parentheses are optional; the shell treats this comma-separated list as an array with or without them.

2. In the PowerShell window, type the following command and press ENTER:

```
Get-ADUser -Filter 'Office -eq "Bellevue"' -Properties Office
```

**Results:** After this exercise, you should understand that many short scripts can be converted into simple one-liner commands that are efficient and require less memory.

## Lab Review

1. Name three different keywords that allow you to do something in a loop.  
**Answer:** for, foreach, and while
2. What keyword should you use when you need to compare one value against many other individual values?  
**Answer:** switch
3. What makes a PowerShell one-liner so powerful?  
**Answer:** The fact that it can process a limitless amount of data and make changes to that data without many commands and without a large memory footprint.
4. What is the name of the top-most scope?  
**Answer:** Global

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab C: Error Trapping and Handling

## Before You Begin

1. Start the **LON-DC1** virtual machine. Wait until the boot process is complete.
2. Start the **LON-SVR1** virtual machine and log on with the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
3. On LON-SVR1, open the **Windows PowerShell ISE**. All PowerShell commands will be run within the Windows PowerShell ISE program.
4. In the PowerShell window, type the following command and press ENTER:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

If you receive a prompt to confirm execution policy change, click **Yes**.

5. In the PowerShell window, type the following command and press ENTER:

```
Set-Location E:\Mod11\Labfiles
```

## Exercise 1: Retrieving Error Information

### ► Task 1: Identify the Error Action Preference settings

1. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference
```

2. In the PowerShell window, type the following command and press ENTER:

```
Get-Help about_preference_variables
```

3. In the PowerShell window, type the following command and press ENTER:

```
Get-Help about_CommonParameters
```

### ► Task 2: Control nonterminating error action in scope

1. In the PowerShell window, type the following command and press ENTER:

```
Set-Location E:\Mod11\Labfiles
```

2. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference = 'Continue'
```

3. In the PowerShell window, type the following command and press ENTER:

```
./ErrorActionPreference.ps1 -FilePath Servers.txt
```

4. In the PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
./ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

5. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference = 'SilentlyContinue'
```

6. In the PowerShell window, type the following command and press ENTER:

```
./ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

7. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference = 'Inquire'
```

8. In the PowerShell window, type the following command and press ENTER:

```
./ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

9. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference = 'Stop'
```

10. In the PowerShell window, type the following command and press ENTER:

```
./ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

11. Close the Windows PowerShell window.

► **Task 3: Control error action per command**

1. Open the Windows PowerShell ISE again and click the **File** menu, click on the **Open** menu item, and navigate to and select **E:\Mod11\Labfiles\ErrorActionPreference.ps1**.
2. In the PowerShell window, type the following command and press ENTER:

```
$ErrorActionPreference = 'Continue'
```

3. Edit line 6 of **ErrorActionPreference.ps1** to be the following and save the file:

```
[string[]]$servers = Get-Content $FilePath -ErrorAction 'SilentlyContinue'
```

4. In the PowerShell window, type the following command and press ENTER:

```
.\ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

5. Edit line 6 of **ErrorActionPreference.ps1** to be the following and save the file:

```
[string[]]$servers = Get-Content $FilePath -ErrorAction 'Inquire'
```

6. In the PowerShell window, type the following command and press ENTER:

```
.\ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```

7. Edit line 6 of **ErrorActionPreference.ps1** to be the following and save the file:

```
[string[]]$servers = Get-Content $FilePath -ErrorAction 'Stop'
```

8. In the PowerShell window, type the following command and press ENTER  

```
.\ErrorActionPreference.ps1 -FilePath DoesNotExist.txt
```
9. Edit line 6 of **ErrorActionPreference.ps1** to be the following and save the file:  

```
[string[]]$servers = Get-Content $FilePath$err
```
10. Do not close the file in the script editor.

► **Task 4: Review errors globally**

1. In the PowerShell window, type the following command and press ENTER:

```
Set-Location E:\Mod11\Labfiles
```

2. In the PowerShell window, type the following command and press ENTER:  

```
$ErrorActionPreference = 'Continue'
```
3. In the PowerShell window, type the following command and press ENTER:  

```
./ErrorActionPreference.ps1 -FilePath ServersWithErrors.txt
```

4. In the PowerShell window, type the following command and press ENTER:  

```
$Error
```
5. In the PowerShell window, type the following command and press ENTER:  

```
$Error[0] | get-member
```

6. Close the Windows PowerShell window.

► **Task 5: Review errors by command**

1. Open the Windows PowerShell ISE and in the script pane edit line 12 of **ErrorActionPreference.ps1** to be the following and save the file:

```
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem -Computername $server -  
ErrorVariable global:ComputerSystemError
```

2. In the PowerShell window, type the following command and press ENTER:  

```
.\ErrorActionPreference.ps1 -FilePath ServersWithErrors.txt
```
3. In the PowerShell window, type the following command and press ENTER:  

```
$ComputerSystemError | Format-List *
```
4. In the PowerShell window, type the following command and press ENTER:  

```
$ComputerSystemError | Get-Member
```

5. In the PowerShell window, type the following command and press ENTER:

```
Remove-Variable ComputerSystemError
```

6. In the script pane, edit line 12 of **ErrorActionPreference.ps1** to be the following and save the file:

```
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem -Computername $server -  
ErrorVariable +global:ComputerSystemError
```

7. In the PowerShell window, type the following command and press ENTER:

```
.\ErrorActionPreference.ps1 -FilePath ServersWithErrors.txt.
```

8. In the PowerShell window, type the following command and press ENTER:

```
$ComputerSystemError | Format-List *
```

9. In the PowerShell window, type the following command and press ENTER:

```
$ComputerSystemError | Get-Member
```

10. In the script pane, edit line 12 of **ErrorActionPreference.ps1** to be the following and save the file:

```
$ComputerSystem = Get-WmiObject -Class Win32_ComputerSystem -Computername $server.
```

**Results:** After this exercise, you should have identified terminating and nonterminating errors, controlled the Windows PowerShell runtime's Error Action Preference both in scope and by command, and examined where error information is available.

## Exercise 2: Handling Errors

### ► Task 1: Handle scoped errors

- In the Windows PowerShell ISE, click the **File** menu, click on the **Open** menu item, and navigate to and select **E:\Mod11\Labfiles\UsingTrap.ps1**.
- In the PowerShell window, type the following command and press ENTER:

```
./UsingTrap.ps1 -FilePath ServersWithErrors.txt
```

- In the script pane, edit the trap statement to include **\$\_ | get-member** to find additional information that can be used to enhance an error message.
- Moving the trap statement above the foreach statement on line 38 stops the foreach loop at the first error. Moving the trap statement under the if statement at line 49 does not catch any ping errors but catches errors from within the **InventoryServer** function.

### ► Task 2: Handle error prone sections of code

- In the Windows PowerShell ISE, click the **File** menu, click on the **Open** menu item, and navigate to and select **E:\Mod11\Labfiles\AddingTryCatch.ps1**.
- In the script pane, edit the if statement on lines 42 to 45 to be the following and save the file:

```
try{  
    if ( $ping.Send("$server").Status -eq 'Success')  
    {
```

```
        InventoryServer($server)
    }
}
Catch {
    Write-Error "Unable to ping $server"
    Write-Error "The error is located on line number: $(
($_.InvocationInfo.ScriptLineNumber)"
    Write-Error "$($_.InvocationInfo.Line.Trim())"
}
```

3. From the command pane, enter the following and press ENTER:

```
./AddingTryCatch.ps1 -FilePath ServersWithErrors.txt
```

#### ► Task 3: Clean up after error prone code

1. In the script pane, after the closing brace in the catch statement, add the following and save the file:

```
finally {
    Write-Host "Finished with $server"
}
```

**Question:** What type of scenarios would a **finally** block be useful for?

**Answer:** The finally block is useful for closing out database connections, disposing of COM objects, unregistering events tied to unmanaged code (via the Win32 API), and cleaning up after any long-running process with potential errors.

#### ► Task 4: Catch specific types of errors

1. In the script pane, change the first line of the catch statement from:

```
catch {
```

to:

```
catch [System.InvalidOperationException] {
```

and save the script.

2. In the command pane, enter the following command and press ENTER:

```
./AddingTryCatch.ps1 -FilePath ServersWithErrors.txt.
```

3. Click the **File** menu, click on the open item, and navigate to and select

**E:\Mod11\Labfiles\UsingTrap.ps1**.

4. In the script pane, edit the trap statement from:

```
trap {
```

to:

```
trap [System.InvalidOperationException] {
```

and save the file.

5. In the command pane, enter the following command and press ENTER:

```
./UsingTrap.ps1 -FilePath ServersWithErrors.txt.
```

- Save both files and close the Windows PowerShell ISE.

**Results:** After this exercise, you should have an understanding of how the Trap and Try...Catch...Finally constructs can catch errors while a script is running. You should also have started to identify potential trouble spots in scripts as areas to focus your error handling efforts.

## Exercise 3: Integrating Error Handling

### ► Task 1: Set up the shell environment

- On the **Start** menu click **All Programs**, click **Accessories**, click **Windows PowerShell**, right-click **Windows PowerShell**, and choose **Run as Administrator**.
- In the PowerShell window, type the following command and press ENTER:

```
Set-ExecutionPolicy 'RemoteSigned' -Scope CurrentUser
```

- Click **Yes** in the Execution Policy Change dialogue when prompted to confirm the change.
- In the PowerShell window, type the following command and press ENTER:

```
Set-Location E:\Mod11\Labfiles
```

### ► Task 2: Create new Event Source for the Windows PowerShell Event Log

- In the PowerShell window, type the following command and press ENTER:

```
New-EventLog -Source AuditScript -LogName 'Windows PowerShell'
```

### ► Task 3: Write error messages to the Windows PowerShell Event Log

- In the PowerShell window, type the following command and press ENTER:

```
Write-EventLog -LogName 'Windows PowerShell' -Source 'AuditScript' -Message 'This is a test' -EventID 3030
```

- In the PowerShell window, type the following command and press ENTER:

```
Show-EventLog
```

- In the event viewer, verify that your log entry was written to the Windows PowerShell event log under **Event Viewer > Applications and Service Logs > Microsoft > Windows PowerShell**.
- Close Windows PowerShell.

### ► Task 4: Add a top level Trap function to catch unexpected errors and log them to the event log

- On the **Start** menu click **All Programs**, click **Accessories**, click **Windows PowerShell**, and click **Windows PowerShell ISE**.
- In the Windows PowerShell ISE, click the **File** menu, click on the **Open** menu item, and navigate to and select **E:\Mod11\Labfiles\AuditScript.ps1**.

3. In the script pane, between the param block and the first function, enter the following and save the script:

```
trap {  
    $message = @'  
Unhandled error in the script.  
    from $($_.InvocationInfo.ScriptName)  
    at line :$($_.InvocationInfo.ScriptLineNumber)  
    by: $($_.InvocationInfo.InvocationName)  
    Message: $($_.exception.message)  
'@  
    Write-EventLog -LogName "Windows PowerShell" -Source "AuditScript" -EventID  
3030 -EntryType Error -Message $message }
```

► **Task 5: Add localized Try...Catch error handling to potential trouble spots, and log the errors to the event log**

1. In the script pane, edit the script to match **AuditScriptFinal.ps1**.
2. In the command pane, type the following and press ENTER:

```
./AuditScript.ps1 -FilePath AuditTest0.txt
```

3. In the command pane, type the following and press ENTER:

```
./AuditScript.ps1 -FilePath AuditTest1.txt
```

4. In the command pane, type the following and press ENTER:

```
./AuditScript.ps1 -FilePath AuditTest2.txt
```

5. In the command pane, type the following and press ENTER:

```
Show-EventLog
```

6. Verify that errors were written to the event log from the script.

7. In the command pane, type the following and press ENTER:

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest0.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest1.txt.xml)
```

8. In the command pane, type the following and press ENTER:

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest1.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest2.txt.xml)
```

9. In the command pane, type the following and press ENTER:

```
Compare-Object -ReferenceObject (Import-CliXML AuditLog-AuditTest0.txt.xml) -  
DifferenceObject (Import-CliXML AuditLog-AuditTest2.txt.xml)
```

10. Close the Windows PowerShell ISE.

**Results:** After this exercise, you should have created an event source in the Windows PowerShell event log, created a top-level trap to catch unexpected errors and log them, and wrapped potentially error prone sections of script in a Try...Catch construct.

## Lab Review

1. How can you find out what members the error records have?

**Answer:** \$host.EnterNestedPrompt() to your trap or catch statement (or set a breakpoint) and work interactively with the error record.

2. What type of information are the error records providing you?

**Answer:** What the error was, where the error was, and what the script state was at the time of the error.

3. What type of tasks might you need to do regardless of whether a task succeeds or fails?

**Answer:** Close database connections, remove references to COM objects, and log success or failures to databases.

4. How would you find any cmdlets that work with the event log?

**Answer:** Get-Command \*EventLog\*

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab D: Debugging a Script

## Before You Begin

1. Start the **LON-DC1** virtual machine, and then log on by using the following credentials:
  - Username: **CONTOSO\administrator**
  - Password: **Pa\$\$w0rd**
2. Open a Windows PowerShell session.

### Exercise 1: Debugging from the Windows PowerShell Console

► **Task 1: Use \$DebugPreference to enable debugging statements**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-Location E:\Mod11\Labfiles
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
.\EvaluatePingTimes.ps1 -Path .\PingTestResults.xml -FirstRun
```

3. In the Windows PowerShell window, type the following command and press ENTER:

```
$DebugPreference = 'Continue'
```

4. In the Windows PowerShell window, type the following command and press ENTER:

```
.\EvaluatePingTimes.ps1 -Path .\PingTestResults.xml -FirstRun
```

► **Task 2: Use Write-Debug to provide information as the script progresses**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Notepad ./CalculatePerfMonStats.ps1
```

2. Add **Write-Debug** statements inside functions, inside loops, and in other areas where variables change or are assigned, such as on line 20:

```
Write-Debug $SortedValues[$MiddleOfResults]
```

3. Save **CalculatePerfMonStats.ps1**.

4. In the Windows PowerShell window, type the following command and press ENTER:

```
.\CalculatePerfMonStats.ps1 -Path .\PerfmonData.csv
```

► **Task 3: Use Set-PSDebug to trace the progress through the script**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
$DebugPreference = 'SilentlyContinue'
```

2. In the Windows PowerShell window, type the following command and press ENTER:

MCT USE ONLY. STUDENT USE PROHIBITED

```
Set-PSDebug -Trace 1
```

3. In the Windows PowerShell window, type the following command and press ENTER:

```
./EvaluatePingTimes.ps1 -Path PingTestResults.xml -FirstRun
```

4. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-PSDebug -Trace 2
```

5. In the Windows PowerShell window, type the following command and press ENTER:

```
.\EvaluatePingTimes.ps1 -Path .\PingTestResults.xml -FirstRun
```

6. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-PSDebug -Off
```

► **Task 4: Use Set-PSBreakpoint to break into a script's progress at specified points**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-PSBreakpoint -Line 36, 40, 44, 48 -Script EvaluatePingTimes.ps1
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
.\EvaluatePingTimes.ps1 -Path .\PingTestResults.xml -FirstRun
```

3. In the Windows PowerShell window, type the following command at the nested prompt and press ENTER:

```
$_ | Format-List
```

► **Task 5: Use Get-PSBreakpoint and Remove-PSBreakpoint to manage breakpoints**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-PSBreakpoint
```

2. Note the ID of one of the breakpoints for the next command

3. In the Windows PowerShell window, type the following command and press ENTER:

```
Remove-PSBreakpoint -ID <ID of one of the existing breakpoints>
```

4. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-PSBreakpoint
```

5. In the Windows PowerShell window, type the following command and press ENTER:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

► **Task 6: Use Set-PSBreakpoint to break into a script if certain conditions are met**

1. In the Windows PowerShell window, type the following command and press ENTER:

```
Set-PSBreakpoint -Line 44 -Script .\EvaluatePingTimes.ps1 -Action { if  
($_.RoundtripTime -eq 35) {break} else {continue}}
```

2. In the Windows PowerShell window, type the following command and press ENTER:

```
.\EvaluatePingTimes.ps1 -Path .\PingTestResults.xml -FirstRun
```

3. In the Windows PowerShell window, type the following command at the nested prompt and press ENTER:

```
$_.RoundtripTime
```

4. In the Windows PowerShell window, type the following command at the nested prompt and press ENTER:

```
exit
```

5. Close the Windows PowerShell Prompt.

**Results:** After this exercise, you should have an understanding of how the \$DebugPreference automatic variable controls debugging messages, know how to step through troublesome scripts and track how things are changing, and set breakpoints in a script to further troubleshoot script performance.

## Exercise 2: Debugging Using the Windows PowerShell Integrated Scripting Environment (ISE)

### ► Task 1: Set a breakpoint

1. In the Windows PowerShell ISE script pane with the file **E:\Mod11\Labfiles\CalculatePerfMonStats.ps1** open and loaded into the ISE, set the cursor on line 33. Select **Toggle Breakpoint** from the **Debug** menu.
2. Right-click on line 25 in the script pane. Choose **Toggle Breakpoint**.
3. Set the cursor on line 44 in the script pane. Press F9.
4. In the Windows PowerShell command window, type the following command and press ENTER: (Ensure the set-location value is still E:\Mod11\Labfiles.)

```
Get-PSBreakpoint
```

5. In the command window, type the following command and press ENTER:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

### ► Task 2: Step through the script

1. In the Windows PowerShell command window, type the following command and press ENTER:

```
Set-PSBreakpoint -Command CreateStatistics -Script CalculatePerfMonStats.ps1 -Action  
{if ($CounterValue -like 'Memory\Pages/sec') {break;} else {continue}}
```

2. In the command window, type the following command and press ENTER:

```
./CalculatePerfMonStats.ps1 -Path PerfmonData.csv
```

3. When the script breaks, press F11. Press F11 again to step into the **CalculateMedian** function.
4. Press F10 to step over the first line of the **CalculateMedian** function.
5. Press SHIFT+F11 to step out of the **CalculateMedian** function.
6. When the script breaks, in the command pane, type the following command and press ENTER:

```
$Stats | Format-List
```

7. When the script breaks, in the command pane, type the following command and press ENTER:  

```
exit
```
8. Close the Windows PowerShell ISE.

**Results:** After this exercise, you should have used the integrated debugging features in the Windows PowerShell ISE to debug a script by setting breakpoints and stepping through the script.

MCT USE ONLY. STUDENT USE PROHIBITED

## Lab Review

1. What are the key variables in your script, and when are they used?

**Answer:** Identifying the key variables provides a starting place for setting breakpoints on variables or as places to surround with Write-Debug statements.

2. Are your commands getting the correct inputs?

**Answer:** Using the debugging techniques, we can set breakpoints, use tracing, and step through our scripts to verify that what is coming into a script, function, or cmdlet is what we expect.

3. Are there hard coded paths, filenames, IP addresses, or other resources that may vary depending on where the script is run?

**Answer:** Having hard-coded values in a script make it more brittle, requiring that you change the script to run in a different environment. If during debugging, you see hard-coded values, consider moving them to a parameter.

4. Are the variable names spelled correctly?

**Answer:** Use Set-PSDebug –Strict to verify that you are not accessing variables that have not been initialized.

5. Are the variable names easy to distinguish from each other?

**Answer:** If there are similarly spelled variables that are hard to distinguish, spelling mistakes and transposition are common errors.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab E: Modularization

## Before You Begin

1. Start the **LON-DC1**, **LON-SVR1**, **LON-SVR2**, and **LON-CLI1** virtual machines. Wait until the boot process is complete.
2. Log on to the **LON-CLI1** virtual machine with the following credentials:
  - a. Username: **CONTOSO\administrator**
  - b. Password: **Pa\$\$w0rd**
3. Open the **Windows PowerShell ISE**. All PowerShell commands are run within the Windows PowerShell ISE program.
4. Create a text file containing the names of all computers in the domain, one on each line, and save it as **C:\users\Administrator\documents\computers.txt**.

## Exercise 1: Generating an Inventory Audit Report

### ► Task 1: Create an inventory gathering function

1. In the PowerShell ISE window, type the following command into the command pane and press ENTER:

```
Get-Help New-Object -Full
```

2. In the PowerShell ISE window, type the following command into the command pane and press ENTER:

```
Get-Help about_functions
```

3. In the PowerShell ISE window, type the following into the script pane, pressing ENTER at the end of every line:

```
Function Get-Inventory {  
    PROCESS {  
        $bios = Get-WmiObject -Class Win32_BIOS -ComputerName $_  
        $os = Get-WmiObject -Class Win32_OperatingSystem  
            -ComputerName $_  
        $processor = Get-WmiObject -Class Win32_Processor  
            -ComputerName $_  
        $obj = New-Object -TypeName PSObject  
        $obj | Add-Member NoteProperty ComputerName $_  
        $obj | Add-Member NoteProperty BiosSerialNumber  
            ($bios.serialnumber)  
        $obj | Add-Member NoteProperty OSVersion  
            ($os.caption)  
        $obj | Add-Member NoteProperty SPVersion  
            ($os.servicepackmajorversion)  
        $obj | Add-Member NoteProperty NumProcessorCores  
            ($processor.numberofcores)  
        write-output $obj  
    }  
}  
'LON-DC1','LON-SVR1','LON-SVR2' | Get-Inventory
```

4. Run the commands if it has not already run by pressing Enter in the last step. You can run the commands by highlighting the code you wish to run, right clicking and going to "Run Selection" or by pressing F5. In the resultant diaogue type any text and click **OK**.

**Note:** If you are receiving error messages when running it double check that your formatting is correct as well as the syntax. You could also save this as a script and then run that script by calling it from the Windows PowerShell command line.

► **Task 2: Export the results of the function in three formats: .txt, .csv, and .xml**

1. In the PowerShell ISE window, modify this line:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory
```

to read:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory |  
Out-File inventory.txt
```

2. In the PowerShell ISE window, modify this line:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory |  
Out-File inventory.txt
```

to read:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory |  
Export-Csv inventory.csv
```

3. In the PowerShell ISE window, modify this line:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory |  
Export-Csv inventory.csv
```

to read:

```
'LON-DC1', 'LON-SVR1', 'LON-SVR2' | Get-Inventory |  
Export-Clixml inventory.xml
```

**Results:** After this exercise, you should be able to create a pipeline function (also known as a filter) that processes information passed in from the pipeline and uses it to retrieve other information. You should also be able to combine multiple objects into one object and export data from a function in multiple formats.

## Lab Review

1. What parameter attributes are used to indicate that a parameter accepts pipeline input?

**Answer:** ValueFromPipeline and ValueFromPipelineByPropertyName

2. What is the advantage of using an advanced function instead of a regular function?

**Answer:** There are many advantages. Advanced functions support help documentation, parameter and cmdlet attributes so that PowerShell can handle many things automatically like parameter validation and WhatIf, it's easier to accept pipeline input in advanced functions, advanced functions support parameter sets are a few examples of how advanced functions offer advantages over regular functions.

MCT USE ONLY. STUDENT USE PROHIBITED

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