

Robert van den Nieuwendijk

Learning PowerCLI

Second Edition

Learn to leverage the power of PowerCLI to automate your VMware vSphere environment with ease



Packt>

Table of Contents

[Learning PowerCLI Second Edition](#)

[Credits](#)

[About the Author](#)

[About the Reviewer](#)

[www.PacktPub.com](#)

[Why subscribe?](#)

[Customer Feedback](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introduction to PowerCLI](#)

[Downloading and installing PowerCLI](#)

[Requirements for using PowerCLI 6.5 Release 1](#)

[Downloading PowerCLI 6.5 Release 1](#)

[Installing PowerCLI](#)

[Participating in the VMware Customer Improvement Program](#)

[Modifying the PowerShell execution policy](#)

[Creating a PowerShell profile](#)

[Connecting and disconnecting servers](#)

[Connecting to a server](#)

[Connecting to multiple servers](#)

[Suppressing certificate warnings](#)

[Disconnecting from a server](#)

[Retrieving the PowerCLI configuration](#)

[Using the credential store](#)

[Retrieving a list of all of your virtual machines](#)

[Suppressing deprecated warnings](#)

[Using wildcard characters](#)

[Filtering objects](#)

[Using comparison operators](#)

[Using aliases](#)

[Retrieving a list of all of your hosts](#)

[Displaying the output in a grid view](#)

[Summary](#)

[2. Learning Basic PowerCLI Concepts](#)

[Using the Get-Command, Get-Help, and Get-Member cmdlets](#)

[Using Get-Command](#)

[Using Get-VICommand](#)

[Using Get-Help](#)

[Using Get-PowerCLIHelp](#)

[Using Get-PowerCLICommunity](#)

[Using Get-Member](#)

[Using providers and PSDrives](#)

[Using providers](#)

[Using PSDrives](#)

[Using the PowerCLI Inventory Provider](#)

[Using the PowerCLI Datastore Provider](#)

[Copying files between a datastore and your PC](#)

[Using arrays and hash tables](#)

[Creating calculated properties](#)

[Using raw API objects with ExtensionData or Get-View](#)

[Using the ExtensionData property](#)

[Using the Get-View cmdlet](#)

[Using managed object references](#)

[Using the Get-VIObjectByVIView cmdlet](#)

[Extending PowerCLI objects with the New-VIProperty cmdlet](#)

[Working with vSphere folders](#)

[Summary](#)

[3. Working with Objects in PowerShell](#)

[Using objects, properties, and methods](#)

[Using methods](#)

[Expanding variables and subexpressions in strings](#)

[When will a string be expanded?](#)

[Expanding a string when it is used](#)

[Using here-strings](#)

[Using the pipeline](#)

[Using the ByValue parameter binding](#)

[Using the ByPropertyName parameter binding](#)

[Using the PowerShell object cmdlets](#)

[Using the Select-Object cmdlet](#)

[Using the Where-Object cmdlet](#)

[Using the ForEach-Object cmdlet](#)

[Using the Sort-Object cmdlet](#)

[Using the Measure-Object cmdlet](#)

[Rounding a value](#)

[Using the Group-Object cmdlet](#)

[Using the Compare-Object cmdlet](#)

[Using the Tee-Object cmdlet](#)

[Creating your own objects](#)

[Using the New-Object cmdlet](#)

[Using a hash table to create an object](#)

[Creating objects using the Select-Object cmdlet](#)

[Creating objects using \[pscustomobject\]](#)

[Adding properties to an object with Add-Member](#)

[Using COM objects](#)

[Summary](#)

[4. Managing vSphere Hosts with PowerCLI](#)

[Adding hosts to a VMware vCenter Server](#)

[Creating a data center](#)

[Creating a cluster](#)

[Adding a host](#)

[Enabling and disabling maintenance mode](#)

[Working with host profiles](#)

[Creating a host profile](#)

[Attaching the host profile to a cluster or a host](#)

[Testing the host profile for compliance](#)

[Applying a host profile to a host or cluster](#)

[Using host profile answer files](#)

[Exporting a host profile](#)

[Importing a host profile](#)

[Working with host services](#)

[Retrieving information about host services](#)

[Starting a host service](#)

[Stopping a host service](#)

[Restarting a host service](#)

[Modifying the startup policy of a host service](#)

[Configuring the host firewall](#)

[Getting the host firewall default policy](#)

[Modifying the host firewall default policy](#)

[Getting the host firewall exceptions](#)

[Modifying a host firewall exception](#)

[Using vSphere Image Builder and Auto Deploy](#)

[Using Image Builder](#)

[Adding ESXi software depots to your PowerCLI session](#)

[Retrieving the ESXi software depots added to your PowerCLI session](#)

[Retrieving the image profiles in your PowerCLI session](#)

[Creating image profiles](#)

[Retrieving VIB objects from all of the connected depots](#)

[Adding VIBs to an image profile or updating existing VIBs](#)

[Exporting an image profile to an ISO or ZIP file](#)

[Configuring Auto Deploy](#)

[Creating deploy rules](#)

[Adding deploy rules to a ruleset](#)

[Retrieving deploy rulesets](#)

[Adding host profiles to a deploy ruleset](#)

[Using esxcli from PowerCLI](#)

[Removing hosts from a VMware vCenter Server](#)

[Summary](#)

[5. Managing Virtual Machines with PowerCLI](#)

[Creating virtual machines](#)

[Creating virtual machines from scratch](#)

[Creating virtual machines from templates](#)

[Cloning virtual machines](#)

[Registering virtual machines](#)

[Using OS customization specifications](#)

[Importing OVF or OVA packages](#)

[Retrieving the required properties](#)

[Assigning values to the required properties](#)

[Importing the vMA OVF file](#)

[Starting and stopping virtual machines](#)

[Starting virtual machines](#)

[Suspending virtual machines](#)

[Shutting down the virtual machine's guest operating systems](#)

[Stopping virtual machines](#)

[Modifying the settings of virtual machines](#)

[Using the VMware vSphere API to modify virtual machine settings](#)

[Adding devices to a virtual machine](#)

[Adding a hard disk](#)

[Adding a SCSI controller](#)

[Adding a network adapter](#)

[Adding a floppy drive](#)

[Adding a CD drive](#)

[Modifying devices added to a virtual machine](#)

[Modifying a hard disk](#)

[Moving a hard disk to another datastore](#)

[Modifying a SCSI controller](#)

[Modifying a network adapter](#)

[Modifying a floppy drive](#)

[Modifying a CD drive](#)

[Removing devices from a virtual machine](#)

[Removing a hard disk](#)

[Removing a network adapter](#)

[Removing a floppy drive](#)

[Removing a CD drive](#)

[Converting virtual machines into templates](#)

[Converting templates into virtual machines](#)

[Modifying the name of a template](#)

[Removing templates](#)

[Moving virtual machines to another folder, host, cluster, resource pool, or datastore](#)

[Updating VMware Tools](#)

[Using the Update-Tools cmdlet](#)

[Enabling the Check and upgrade VMware Tools before each power on checkbox](#)

[Upgrading virtual machine compatibility](#)

[Using snapshots](#)

[Creating snapshots](#)

[Retrieving snapshots](#)

[Reverting to a snapshot](#)

[Modifying snapshots](#)

[Removing snapshots](#)

[Running commands in the guest OS](#)

[Configuring Fault Tolerance](#)

[Turning Fault Tolerance on](#)

[Turning Fault Tolerance off](#)

[Opening the console of virtual machines](#)

[Removing virtual machines](#)

[Using tags](#)

[Managing tag categories](#)

[Creating tag categories](#)

[Retrieving tag categories](#)

[Modifying tag categories](#)

[Removing tag categories](#)

[Managing tags](#)

[Creating tags](#)

[Retrieving tags](#)

[Modifying tags](#)

[Removing tags](#)

[Managing tag assignments](#)

[Creating tag assignments](#)

[Retrieving tag assignments](#)

[Retrieving virtual machines by tag](#)

[Removing tag assignments](#)

[Converting custom attributes and annotations to tags](#)

[Creating tag categories from custom attributes](#)

[Creating tags from annotations](#)

[Summary](#)

[6. Managing Virtual Networks with PowerCLI](#)

[Using vSphere Standard Switches](#)

[Creating vSphere Standard Switches](#)

[Configuring vSphere Standard Switches](#)

[Adding network adapters to a switch](#)

[Removing vSphere Standard Switches](#)

[Using host network adapters](#)

[Creating host network adapters](#)

[Retrieving host network adapters](#)

[Configuring host network adapters](#)

[Configuring network speed and duplex setting](#)

[Configuring the management network](#)

[Configuring vMotion](#)

[Removing host network adapters](#)

[Configuring NIC teaming](#)

[Using standard port groups](#)

[Creating standard port groups](#)

[Configuring standard port groups](#)

[Removing standard port groups](#)

[Using vSphere Distributed Switches](#)

[Creating vSphere Distributed Switches](#)

[Creating a new vSphere Distributed Switch from scratch](#)

[Cloning a vSphere Distributed Switch](#)

[Creating a vSphere Distributed Switch from an export](#)

[Retrieving vSphere Distributed Switches](#)

[Configuring vSphere Distributed Switches](#)

[Rolling back the configuration of a vSphere Distributed Switch](#)

[Importing the configuration of a vSphere Distributed Switch from a backup](#)

[Upgrading a vSphere Distributed Switch](#)

[Adding hosts to vSphere Distributed Switches](#)

[Retrieving hosts connected to vSphere Distributed Switches](#)

[Adding host physical network adapters to a vSphere Distributed Switch](#)

[Removing host physical network adapters from a vSphere Distributed Switch](#)

[Removing hosts from a vSphere Distributed Switch](#)

[Exporting the configuration of vSphere Distributed Switches](#)

[Removing vSphere Distributed Switches](#)

[Using distributed virtual port groups](#)

[Creating distributed virtual port groups](#)

[Creating distributed virtual port groups from a reference group](#)

[Creating distributed virtual port groups from an export](#)

[Retrieving distributed virtual port groups](#)

[Modifying distributed virtual port groups](#)

[Renaming a distributed virtual port group](#)

[Rolling back the configuration of a distributed virtual port group](#)

[Restoring the configuration of a distributed virtual port group](#)

[Configuring network I/O control](#)

[Enabling network I/O control](#)

[Retrieving the network I/O control enabled status](#)

[Disabling network I/O control](#)

[Exporting the configuration of distributed virtual port groups](#)

[Migrating a host network adapter from a standard port group to a distributed port group](#)

[Removing distributed virtual port groups](#)

[Configuring host networking](#)

[Configuring the network of virtual machines](#)

[Setting the IP address](#)

[Setting the DNS server addresses](#)

[Retrieving the network configurations](#)

[Summary](#)

[7. Managing Storage](#)

[Rescanning for new storage devices](#)

[Creating datastores](#)

[Creating NFS datastores](#)

[Getting SCSI LUNs](#)

[Creating VMFS datastores](#)

[Creating software iSCSI VMFS datastores](#)

[Retrieving datastores](#)

[Setting the multipathing policy](#)

[Configuring vmhba paths to an SCSI device](#)

[Retrieving vmhba paths to an SCSI device](#)

[Modifying vmhba paths to an SCSI device](#)

[Working with Raw Device Mappings](#)

[Configuring storage I/O control](#)

[Retrieving Storage I/O Control settings](#)

[Configuring Storage DRS](#)

[Creating a datastore cluster](#)

[Retrieving datastore clusters](#)

[Modifying datastore clusters](#)

[Adding datastores to a datastore cluster](#)

[Retrieving the datastores in a datastore cluster](#)

[Removing datastores from a datastore cluster](#)

[Removing datastore clusters](#)

[Upgrading datastores to VMFS-5](#)

[Removing datastores](#)

Using VMware vSAN

Configuring VMware vSAN networking

Enabling VMware vSAN on vSphere clusters

Retrieving the devices available for VMware vSAN

Creating VMware vSAN disk groups

Retrieving VMware vSAN disk groups

Adding a host SCSI disk to a VMware vSAN disk group

Retrieving the host disks that belong to a VMware vSAN disk group

Removing disks from a VMware vSAN disk group

Removing VMware vSAN disk groups

Using vSphere storage policy-based management

Retrieving storage capabilities

Using tags to define storage capabilities

Creating SPBM rules

Creating SPBM rule sets

Creating SPBM storage policies

Retrieving SPBM storage policies

Modifying SPBM storage policies

Retrieving SPBM compatible storage

Using SPBM to create virtual machines

Retrieving SPBM-related configuration data of clusters, virtual machines, and hard disks

Associating storage policies with virtual machines and hard disks and enabling SPBM on clusters

Exporting SPBM storage policies

Importing SPBM storage policies

Removing SPBM storage policies

Summary

8. Managing High Availability and Clustering

Creating vSphere HA and DRS clusters

Retrieving clusters

Retrieving the HA master or primary hosts

Retrieving cluster configuration issues

Modifying the cluster settings

Configuring enhanced vMotion compatibility (EVC) mode

Disabling HA

Disabling or enabling host monitoring

Enabling VM and application monitoring

Configuring the heartbeat datastore selection policy

Moving hosts to clusters

Moving clusters

Using DRS rules

Creating VM-VM DRS rules

[Creating VM-host DRS rules](#)

[Creating virtual machines DRS groups](#)

[Creating hosts DRS groups](#)

[Retrieving DRS groups](#)

[Modifying DRS groups](#)

[Adding virtual machines to a DRS group](#)

[Removing virtual machines from a DRS group](#)

[Removing DRS groups](#)

[Creating Virtual Machines to Hosts DRS rules](#)

[Retrieving DRS Rules](#)

[Modifying DRS rules](#)

[Removing DRS rules](#)

[Using DRS recommendations](#)

[Using resource pools](#)

[Creating resource pools](#)

[Retrieving resource pools](#)

[Modifying resource pools](#)

[Moving resource pools](#)

[Configuring resource allocation between virtual machines](#)

[Removing resource pools](#)

[Using Distributed Power Management](#)

[Enabling DPM](#)

[Configuring hosts for DPM](#)

[Testing hosts for DPM](#)

[Putting hosts into standby mode](#)

[Starting hosts](#)

[Retrieving the DPM configuration of a cluster](#)

[Disabling DPM](#)

[Removing clusters](#)

[Summary](#)

[9. Managing vCenter Server](#)

[Working with roles and permissions](#)

[Retrieving privileges](#)

[Using roles](#)

[Creating roles](#)

[Retrieving roles](#)

[Modifying roles](#)

[Removing roles](#)

[Using permissions](#)

[Creating permissions](#)

[Retrieving permissions](#)

[Modifying permissions](#)

[Removing permissions](#)

Managing licenses

Adding license keys to the license inventory

Retrieving license keys from the license inventory

Removing license keys from the license inventory

Assigning licenses to hosts

Retrieving assigned licenses

Using the LicenseDataManager

Associating license keys with host containers

Applying the associated license key to all hosts

Retrieving license key associations

Retrieving all of the license key associations to the host containers in your environment

Retrieving the license keys associated with a specific host container

Retrieving the effective license key of a host container

Modifying license key associations

Removing license key associations

Configuring alarms

Retrieving alarm definitions

Modifying alarm definitions

Creating alarm actions

Configuring the vCenter Server mail server and sender settings

Retrieving alarm actions

Removing alarm actions

Creating alarm action triggers

Retrieving alarm action triggers

Removing alarm action triggers

Retrieving events

Summary

10. Patching ESXi Hosts and Upgrading Virtual Machines

Downloading new patches into the Update Manager repository

Retrieving patches in the Update Manager repository

Using baselines and baseline groups

Retrieving baselines

Retrieving patch baselines

Creating patch baselines

Modifying patch baselines

Attaching baselines to inventory objects

Detaching baselines from inventory objects

Removing baselines

Testing inventory objects for compliance with baselines

Retrieving baseline compliance data

Initializing staging of patches

Remediating inventory objects

[Upgrading or patching ESXi hosts](#)

[Upgrading virtual machine hardware](#)

[Summary](#)

[11. Managing VMware vCloud Director and vCloud Air](#)

[Connecting to vCloud Air servers and vCloud Director servers](#)

[Retrieving organizations](#)

[Retrieving organization virtual datacenters](#)

[Retrieving organization networks](#)

[Retrieving vCloud users](#)

[Using vCloud virtual appliances](#)

[Retrieving vApp templates](#)

[Creating vCloud vApps](#)

[Retrieving vCloud vApps](#)

[Starting vCloud vApps](#)

[Stopping vCloud vApps](#)

[Managing vCloud virtual machines](#)

[Creating vCloud virtual machines](#)

[Retrieving vCloud virtual machines](#)

[Starting vCloud virtual machines](#)

[Stopping vCloud virtual machines](#)

[Using the vCloud Director API with Get-CIView](#)

[Removing vCloud virtual machines](#)

[Removing vCloud virtual appliances](#)

[Creating snapshots](#)

[Retrieving snapshots](#)

[Reverting to snapshots](#)

[Removing snapshots](#)

[Disconnecting from vCloud Director servers](#)

[Summary](#)

[12. Using Site Recovery Manager](#)

[Installing SRM](#)

[Connecting to SRM servers](#)

[Downloading and installing the Meadowcroft.SRM module](#)

[Pairing SRM sites](#)

[Retrieving the name of the local vCenter Server](#)

[Retrieving the remote vCenter Server](#)

[Retrieving the SRM user info](#)

[Managing protection groups](#)

[Creating protection groups](#)

[Retrieving protection groups](#)

[Protecting virtual machines](#)

[Retrieving protected virtual machines](#)

[Unprotecting virtual machines](#)

[Managing recovery plans](#)

[Retrieving recovery plans](#)

[Running recovery plans](#)

[Retrieving the historical results of recovery plans](#)

[Disconnecting from SRM servers](#)

[Summary](#)

[13. Using vRealize Operations Manager](#)

[Connecting to vRealize Operations Manager servers](#)

[Retrieving vRealize Operations Manager resource objects](#)

[Using alerts](#)

[Retrieving alert definitions](#)

[Retrieving alert types](#)

[Retrieving alert subtypes](#)

[Modifying alerts](#)

[Retrieving recommendations](#)

[Retrieving statistic keys](#)

[Retrieving statistical data](#)

[Retrieving local user accounts](#)

[Using the vRealize Operations Manager API](#)

[Getting the user roles](#)

[Creating users](#)

[Removing users](#)

[Retrieving solutions](#)

[Retrieving traversalSpecs](#)

[Creating reports](#)

[Retrieving reports](#)

[Disconnecting from vRealize Operations Manager servers](#)

[Summary](#)

[14. Using REST API to manage NSX and vRealize Automation](#)

[Connecting to REST API servers](#)

[Managing NSX logical switches](#)

[Creating NSX logical switches](#)

[Retrieving NSX logical switches](#)

[Removing NSX logical switches](#)

[Managing NSX logical \(distributed\) routers](#)

[Creating NSX logical \(distributed\) routers](#)

[Retrieving NSX logical \(distributed\) routers](#)

[Removing NSX logical \(distributed\) routers](#)

[Managing NSX Edge services gateways](#)

[Retrieving NSX Edge services gateways](#)

[Removing NSX Edge services gateways](#)

[Connecting to vRA servers](#)

[Managing vRA tenants](#)

- [Creating vRA tenants](#)
- [Retrieving vRA tenants](#)
- [Removing vRA tenants](#)
- [Retrieving vRA business groups](#)
- [Managing vRA reservations](#)
 - [Creating vRA reservations](#)
 - [Retrieving vRA reservations](#)
- [Managing vRA machines and applications](#)
 - [Retrieving entitled catalog items](#)
 - [Retrieving a template request for an entitled catalog item](#)
 - [Creating vRA machines](#)
 - [Viewing details of a machine request](#)
 - [Retrieving provisioned resources](#)
- [Summary](#)

[15. Reporting with PowerCLI](#)

- [Retrieving log files](#)
- [Creating log bundles](#)
- [Performance reporting](#)
 - [Retrieving the statistical intervals](#)
 - [Retrieving performance statistics](#)
 - [Retrieving metric IDs](#)
- [Exporting reports to CSV files](#)
- [Generating HTML reports](#)
- [Sending reports by e-mail](#)
- [Reporting the health of your vSphere environment with vCheck](#)
- [Summary](#)

Learning PowerCLI Second Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2014

Second edition: February 2017

Production reference: 1200217

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-801-7

www.packtpub.com

Credits

Author Robert van den Nieuwendijk	Copy Editors Safis Editing Dipti Mankame
Reviewer Kim Bottu	Project Coordinator Shweta H. Birwatkar
Commissioning Editor Vijin Boricha	Proofreader Safis Editing
Acquisition Editor Prachi Bisht	Indexer Pratik Shirodkar
Content Development Editor Abhishek Jadhav	Graphics Kirk D'Penha
Technical Editor Gaurav Suri	Production Coordinator Shantanu N. Zagade

About the Author

Robert van den Nieuwendijk is an IT veteran from the Netherlands with over thirty years of experience in Information Technology. He holds a bachelor degree in software engineering. After working a few years as a programmer of air traffic control and vessel traffic management systems, he started his own company Van den Nieuwendijk Informatica in 1988. Since then he has worked as a freelance systems administrator of OpenVMS, Windows Server, Linux, and VMware vSphere systems, for Dutch governmental organizations and cloud providers. During winter he is also a ski and snowboard instructor at an indoor ski school.

With his background as a programmer, he always tries to make his job easier by writing programs or scripts to perform repeating tasks. In the past, he used the C programming language, OpenVMS DCL, Visual Basic Script and KiXtart to do this. Now, he uses Microsoft PowerShell and VMware PowerCLI for all of his scripting work.

Robert is a frequent contributor and moderator at the VMware VMTN Communities. Since 2012 VMware awarded him the vExpert title for his significant contributions to the community and a willingness to share his expertise with others.

He has a blog at <http://rvdnieuwendijk.com> where he writes mainly about VMware PowerCLI, Microsoft PowerShell, and VMware vSphere.

If you want to get in touch with Robert, then you can find him on Twitter. His username is [@rvdnieuwendijk](https://twitter.com/rvdnieuwendijk).

Robert is also the author of Learning PowerCLI, Packt Publishing.

I would like to thank my wife Ali for supporting me writing this second book.

I also want to thank the people at Packt Publishing for giving me the opportunity to update the Learning PowerCLI book and write this second edition.

About the Reviewer

Kim Bottu is the virtualization engineer in the EMEA region for an international Biglaw firm, where he focuses on virtual datacenter operations, optimization, and design. In his current role, he takes care of the consolidated virtual datacenters in Asia and Europe, and he is the SME for the EMEA Litigation virtual datacenters. He holds the following certifications and honors: VCA-NV, VCP5-DCV, VCP6-DCV, VCAP5-DCD, VCAP6-DCV Design, and TOGAF 9 certified. He has also been named vExpert 2016 and vExpert 2017.

Kim currently lives in Belgium and is a proud dad of a daughter named Zoey. In his spare time you might find him playing with his daughter, reading books, or riding his mountain bike. Kim can be reached at www.vMusketeers.com.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786468018>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Preface

VMware PowerCLI is a command-line automation and scripting tool that provides a Microsoft PowerShell interface to the VMware vSphere and vCloud products. Learning PowerCLI shows you how to install and use PowerCLI to automate the management of your VMware vSphere environment. With lots of examples, this book will teach you how to manage vSphere from the command line and how to create advanced PowerCLI scripts.

What this book covers

[Chapter 1](#), Introduction to PowerCLI, gets you started using PowerCLI. First, you will see how to download and install PowerCLI. Then, you will learn to connect to and disconnect from the vCenter and ESXi servers and retrieve a list of all of your hosts and virtual machines.

[Chapter 2](#), Learning Basic PowerCLI Concepts, introduces the Get-Help, Get-Command, and Get-Member cmdlets. It explains the difference between PowerShell Providers and PSdrives. You will see how you can use the raw vSphere API objects from PowerCLI and how to use the New-VIProperty cmdlet to extend a PowerCLI object.

[Chapter 3](#), Working with Objects in PowerShell, concentrates on objects, properties, and methods. This chapter shows how you can use the pipeline to use the output of one command as the input of another command. You will learn how to use the PowerShell object cmdlets and how to create PowerShell objects.

[Chapter 4](#), Managing vSphere Hosts with PowerCLI, covers the management of the vSphere ESXi servers. You will see how to add hosts to the vCenter server and how to remove them. You will work with host profiles, host services, Image Builder, and Auto Deploy, as well as with the esxcli command and the vSphere CLI commands from PowerCLI.

[Chapter 5](#), Managing Virtual Machines with PowerCLI, examines the lifecycle of virtual machines-from creating to removing them. Creating templates, updating VMware Tools and upgrading virtual hardware, running commands in the guest OS, and configuring fault tolerance are some of the topics discussed in this chapter.

[Chapter 6](#), Managing Virtual Networks with PowerCLI, walks you through vSphere Standard Switches and vSphere Distributed Switches, port groups, and network adapters. It shows you how to configure host networking and how to configure the network of a virtual machine.

[Chapter 7](#), Managing Storage, explores creating and removing datastores and datastore clusters, working with Raw Device Mapping, configuring software iSCSI initiators, Storage I/O Control, and Storage DRS.

[Chapter 8](#), Managing High Availability and Clustering, covers HA and DRS clusters, DRS rules and DRS groups, resource pools, and Distributed Power Management.

[Chapter 9](#), Managing vCenter Server, shows you how to work with privileges, work with roles and permissions, manage licenses, configure alarm definitions, alarm action triggers, and retrieve events.

[Chapter 10](#), Patching ESXi Hosts and Upgrading Virtual Machines, focusses on using VMware vSphere Update Manager to download patches, creating baselines and baseline groups, testing virtual machines and hosts for compliance, staging patches, and remediating inventory objects.

[Chapter 11](#), Managing VMware vCloud Director and vCloud Air, covers connecting to vCloud servers, retrieving organizations, virtual datacenters, organization networks, and users, using vCloud virtual machines and appliances, and using snapshots.

[Chapter 12](#), Using Site Recovery Manager, explores the Meadowcroft.SRM module to

manage SRM protection groups, protecting virtual machines and running recovery plans to migrate or fail-over virtual machines from the protected site to the recovery site.

[Chapter 13](#), Using vRealize Operations Manager, shows you to use alerts, retrieve recommendations, statistical data, solutions, and traversalSpecs, manage local user accounts and user roles and create and retrieve reports.

[Chapter 14](#), Using REST API to Manage NSX and vRealize Automation, walks you through REST APIs with examples from VMware NSX and vRealize Automation using basic authentication and bearer tokens, XML, and JSON.

[Chapter 15](#), Reporting with PowerCLI, concentrates on retrieving log files and log bundles, performance reporting, exporting reports to CSV files, generating HTML reports, sending reports by e-mail, and reporting the health of your vSphere environment with the vCheck script.

What you need for this book

To run the example PowerCLI scripts given in this book, you need the following software:

- VMware PowerCLI
- Microsoft PowerShell
- VMware vCenter Server
- VMware ESXi
- VMware vSphere Update Manager
- VMware vCloud Director
- VMware Site Recovery Manager
- VMware vSphere Replication
- VMware vRealize Operations Manager
- VMware NSX
- VMware vRealize Automation

If you don't have specific software installed, you can use the VMware Hands-on Labs at <https://labs.hol.vmware.com/> to test the scripts.

The scripts in this book are tested using VMware PowerCLI 6.5 Release 1, VMware vCenter Server 6.5, and VMware ESXi 6.5. Microsoft PowerShell and VMware PowerCLI are free. You can download a free 60-day evaluation of VMware vCenter Server and VMware ESXi from the VMware website. It is not possible to modify the settings on the free VMware vSphere Hypervisor using PowerCLI.

Who this book is for

This book is written for VMware vSphere administrators who want to automate their vSphere environment using PowerCLI. It is assumed that you have at least a basic knowledge of VMware vSphere. If you are not a vSphere administrator, but you are interested in learning more about PowerCLI, then this book will also give you some basic knowledge of vSphere.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The script uses the `Get-Cluster` cmdlet to retrieve all the clusters."

A block of code is set as follows:

```
$HostName = '192.168.0.133'
$iSCSITarget = '192.168.0.157'
$VirtualSwitchName = 'vSwitch2'
$NicName = 'vmnic3'
$PortGroupName = 'iSCSI Port group 1'
$ChapType = 'Preferred'
$ChapUser = 'Cluster01User'
$ChapPassword = ' Cluster01Pwd'
$DatastoreName = 'Cluster01_iSCSI01'
```

Any command-line input or output is written as follows:

```
PowerCLI C:\> New-VM -Name VM1 -ResourcePool (Get-Cluster
               -Name Cluster01)
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "If your cluster is incorrectly configured, the vSphere Web Client will show you the issues in the **Summary** tab."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important to us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learning-PowerCLI-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted, and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come

across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Chapter 1. Introduction to PowerCLI

Have you ever had to create 200 virtual machines in a short period of time, change a setting on all of your hosts, or make an advanced report for your boss to show how full the hard disks of your virtual machines are? If you have, you know that performing these tasks using the vSphere web client will take a lot of time. This is where automation can make your job easier. **VMware PowerCLI** is a powerful tool that can perform these tasks and much more. And the best thing is that it is free!

VMware PowerCLI is a **command-line interface (CLI)** distributed as a collection of Microsoft PowerShell modules and snap-ins. **Microsoft PowerShell** is Microsoft's command shell and scripting language, designed with the systems administrator in mind. Microsoft PowerShell is available on every Microsoft Windows server or workstation since Windows Server 2008 R2 and Windows 7. VMware PowerCLI is an extension to Microsoft PowerShell. This means that all of the features of PowerShell can be used in PowerCLI. You can use PowerCLI to automate your vSphere hosts, virtual machines, virtual networks, storage, clusters, vCenter Servers, and more.

In this chapter, you will learn:

- Downloading and installing PowerCLI
- Participating in the VMware Customer Improvement Program
- Modifying the PowerShell execution policy
- Creating a PowerShell profile
- Connecting and disconnecting servers
- Using the credential store
- Retrieving a list of all of your virtual machines
- Retrieving a list of all of your hosts

Downloading and installing PowerCLI

In this section, you will learn how to download and install **PowerCLI 6.5 Release 1**. First, we will list the requirements for PowerCLI 6.5 Release 1. After downloading PowerCLI from the VMware website, we will install PowerCLI on your system.

Requirements for using PowerCLI 6.5 Release 1

You can install VMware PowerCLI 6.5 Release 1, the version used for writing this book, on the following 64-bit operating systems:

- Windows Server 2012 R2
- Windows Server 2008 R2 Service Pack 1
- Windows 10
- Windows 8.1
- Windows 7 Service Pack 1

VMware PowerCLI 6.5 Release 1 is compatible with the following PowerShell versions:

- Microsoft PowerShell 3.0
- Microsoft PowerShell 4.0
- Microsoft PowerShell 5.0

- Microsoft PowerShell 5.1

If you want to work with VMware PowerCLI 6.5 Release 1, make sure that the following software is present on your system:

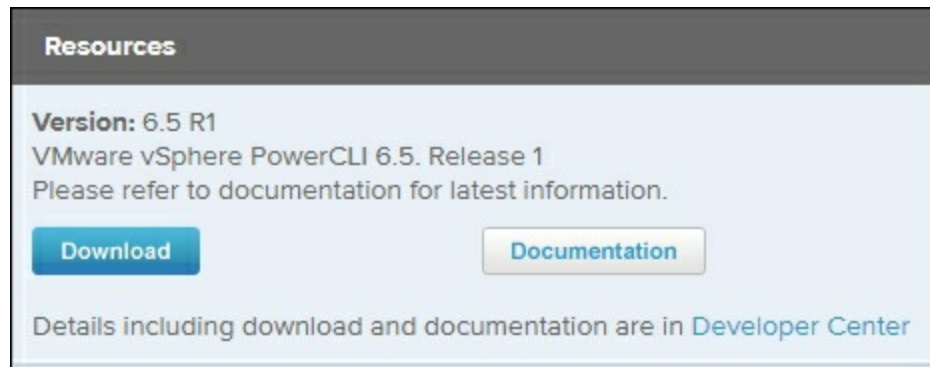
- Microsoft PowerShell 3.0, 4.0, 5.0, or 5.1
- NET Framework 4.5, 4.5.x, 4.6, or 4.6.x

Downloading PowerCLI 6.5 Release 1

Before you can install PowerCLI, you have to download the PowerCLI installer from the VMware website. You will need a **My VMware account** to do this.

Perform the following steps to download PowerCLI:

1. Visit <http://www.vmware.com/go/powercli>. On this page, you will find a **Resources** section.



2. Click on the **Download** button to download PowerCLI.
3. You have to log in with a My VMware account. If you don't have a My VMware account, you can register for free.

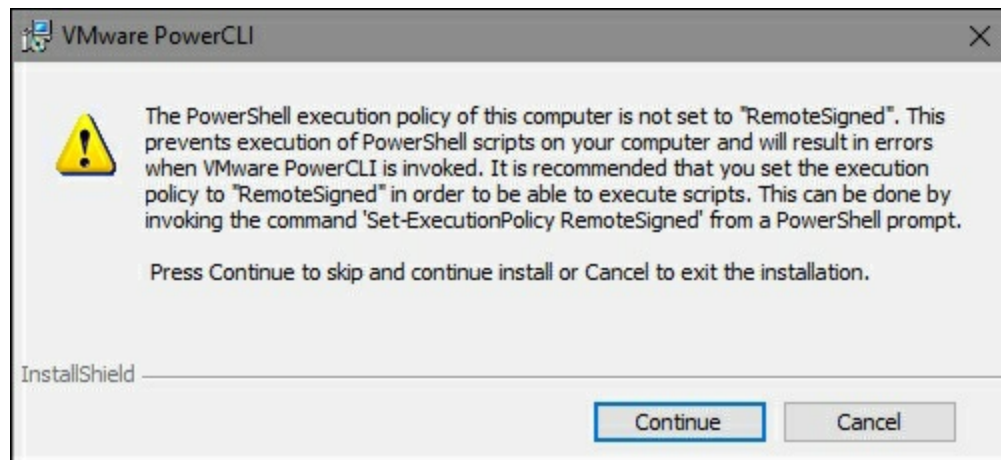
A screenshot of the VMware login and registration page. At the top left is a blue square icon followed by the text 'Log In'. Below this are two input fields: the first is labeled 'Email address or customer number' and the second is empty. To the right of the second input field is a link that says 'Forgot your password?'. Below the input fields is a checkbox labeled 'Remember me'. At the bottom of the login section is a blue button labeled 'Log In'. Below a horizontal line is a green square icon followed by the text 'Register'.

4. After you log in, you will be taken to the **VMware PowerCLI** download page. Click on the **Download Now** button to start downloading PowerCLI.

Installing PowerCLI

Perform the following steps to install PowerCLI:

1. Run the PowerCLI installer that you just downloaded.
2. Click **Yes** in the **User Account Control** window to accept the **Do you want to allow this app to make changes to your device?** option.
3. If the **PowerShell execution policy** on your computer is not set to **RemoteSigned**, you will get a warning that tells you **It is recommended that you set the execution policy to "RemoteSigned" in order to be able to execute scripts**. After the installation of PowerCLI, I will show you how to set the execution policy. Click on **Continue** to continue to the installation of PowerCLI.



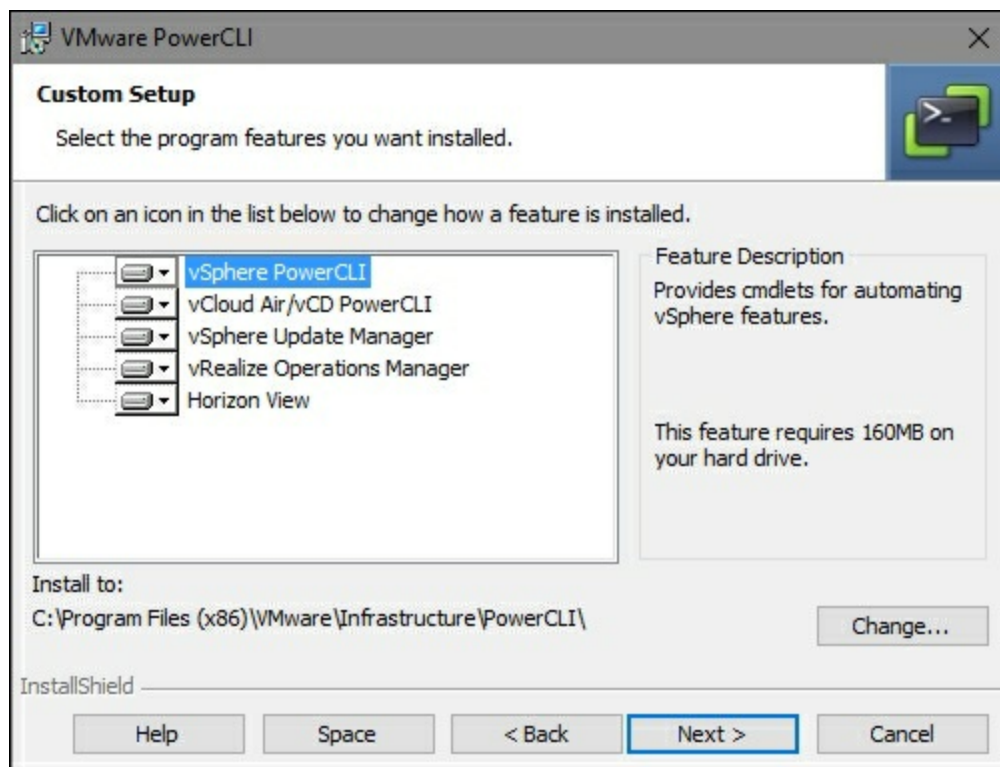
4. Click on **Next >** in the **Welcome to the InstallShield Wizard for VMware PowerCLI** window.



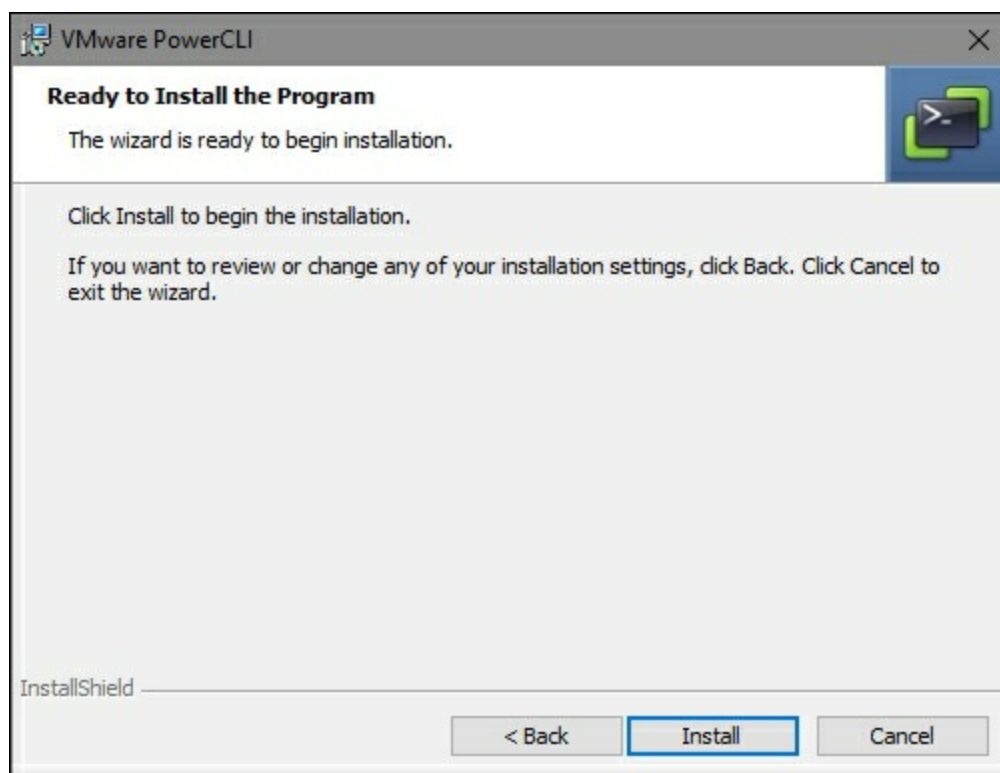
5. Select **I accept the terms in the license agreement** and click on **Next >**.



6. If you are not using **vCloud Air**, **VMware vCloud Director**, **vSphere Update Manager**, **vRealize Operations Manager**, or **Horizon View**, you can click on the little arrow to the left of a feature and select **This feature will not be available**. I recommend installing all of the features, to be able to run the scripts in all of the chapters in this book. If you want, you can change the installation directory by clicking on **Change...**. Click on **Next >**.



7. Click on **Install** to begin the installation.



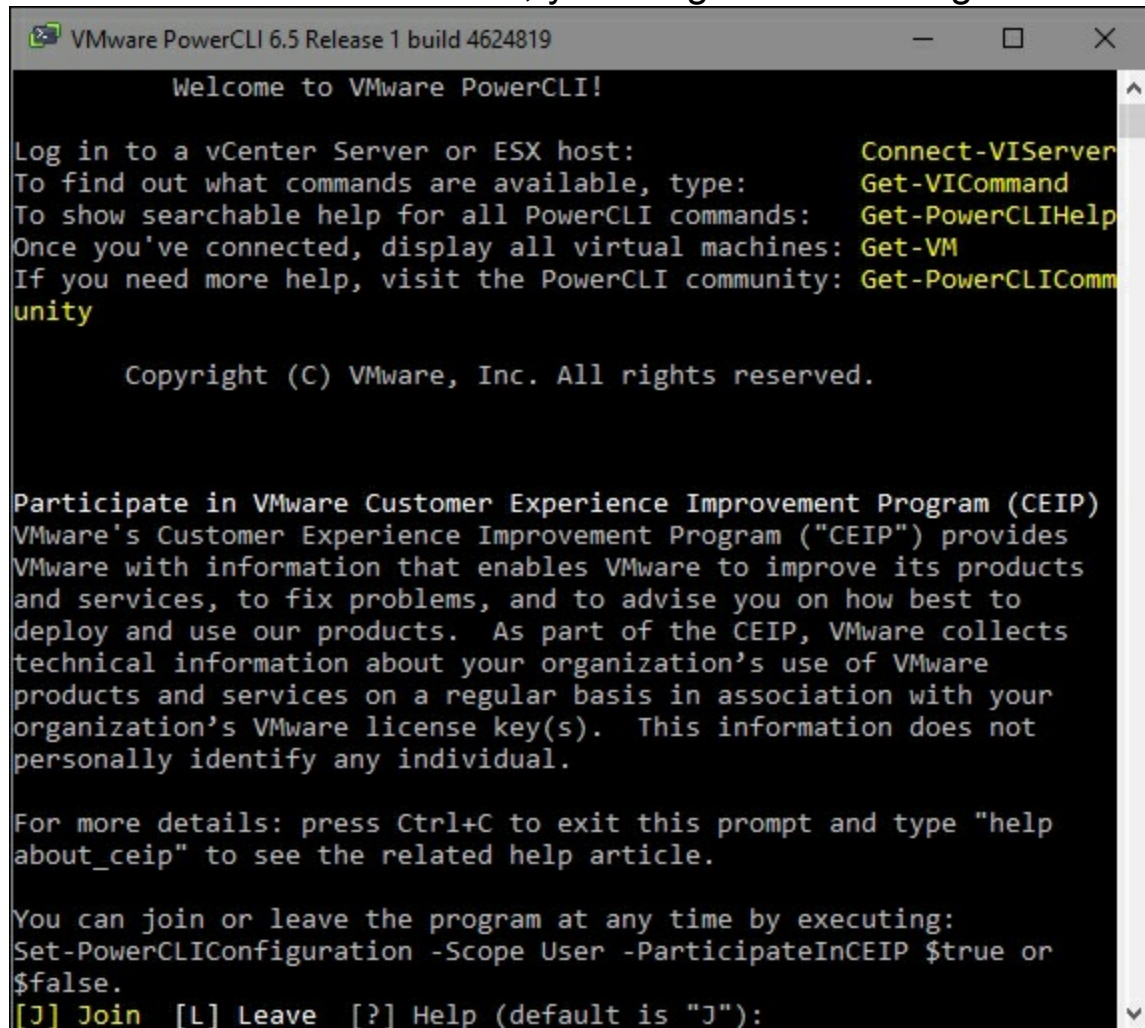
8. Click on **Finish** to exit the installation wizard.



After installing PowerCLI, you will have a **VMware PowerCLI** icon on your desktop. If you installed PowerCLI on a 64-bit computer, you will also have a **VMware PowerCLI (32-Bit)** icon. Some PowerCLI commands only work in the 32-bit version of PowerCLI. So keep both versions.

Participating in the VMware Customer Improvement Program

When you start PowerCLI for the first time, you will get the following screen:



```
VMware PowerCLI 6.5 Release 1 build 4624819

Welcome to VMware PowerCLI!

Log in to a vCenter Server or ESX host:          Connect-VIServer
To find out what commands are available, type:    Get-VICommand
To show searchable help for all PowerCLI commands: Get-PowerCLIHelp
Once you've connected, display all virtual machines: Get-VM
If you need more help, visit the PowerCLI community: Get-PowerCLICommunity

Copyright (C) VMware, Inc. All rights reserved.

Participate in VMware Customer Experience Improvement Program (CEIP)
VMware's Customer Experience Improvement Program ("CEIP") provides
VMware with information that enables VMware to improve its products
and services, to fix problems, and to advise you on how best to
deploy and use our products. As part of the CEIP, VMware collects
technical information about your organization's use of VMware
products and services on a regular basis in association with your
organization's VMware license key(s). This information does not
personally identify any individual.

For more details: press Ctrl+C to exit this prompt and type "help
about_ceip" to see the related help article.

You can join or leave the program at any time by executing:
Set-PowerCLIConfiguration -Scope User -ParticipateInCEIP $true or
$false.
[J] Join [L] Leave [?] Help (default is "J"):
```

You are asked to participate in the **VMware Customer Improvement Program (CEIP)**. Type J to participate in the CEIP or type L to leave.

If you didn't get the text from the preceding screenshot, you may get the following error message:

. : File C:\Program Files (x86)\VMware\Infrastructure\vSphere PowerCLI\Scripts\Initialize-PowerCLIEnvironment.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see about_Execution_Policies at <http://go.microsoft.com/fwlink/?LinkID=135170>.

Then, read the following section, **Modifying the PowerShell execution policy**, to solve this problem.

Modifying the PowerShell execution policy

If this is the first time that you are using Microsoft PowerShell on the computer on which you installed PowerCLI, you have to change the **execution policy** to be able to start PowerCLI.

The Microsoft PowerShell execution policies define when you can run scripts or load configuration files. The possible values for the execution policy are `Restricted`, `AllSigned`, `RemoteSigned`, `Unrestricted`, `Bypass`, and `Undefined`.

Policy	Description
<code>Restricted</code>	This is the default execution policy. It allows you to run commands at the Command Prompt, but disables the execution of scripts. It will also disable the start of PowerCLI.
<code>AllSigned</code>	With the <code>AllSigned</code> execution policy, scripts can run, but they must be signed by a trusted publisher. If you run a script by a publisher that is not trusted yet, you will see a prompt asking whether you trust the publisher of the script.
<code>RemoteSigned</code>	The <code>RemoteSigned</code> execution policy allows you to run scripts that you have written on the local computer. Any script downloaded from the Internet must be signed by a trusted publisher or must be unblocked.
<code>Unrestricted</code>	When the execution policy is set to <code>Unrestricted</code> , unsigned scripts can run. If you run a script that has been downloaded from the Internet, you will get a security warning saying that this script can potentially harm your computer and asking whether you want to run this script.
<code>Bypass</code>	The <code>Bypass</code> execution policy blocks nothing and displays no warnings or prompts. This execution policy is designed for configurations in which a Microsoft PowerShell script is built into a larger application that has its own security model.
<code>Undefined</code>	The <code>Undefined</code> execution policy removes the execution policy from the current scope. If the execution policy in all scopes is <code>Undefined</code> , the effective execution policy is <code>Restricted</code> , which is the default execution policy. The <code>Undefined</code> execution policy will not remove an execution policy that is set in a Group Policy scope.

You can check the current execution policy setting with the following command:

```
PowerCLI C:\> Get-ExecutionPolicy
```

`Get-ExecutionPolicy` is a Microsoft PowerShell **commandlet (cmdlet)**. Cmdlets are commands built into PowerShell or PowerCLI. They follow a verb-noun naming convention. The `get` cmdlets retrieve information about the item that is specified as the noun part of the cmdlet.

Set the execution policy to `RemoteSigned` to be able to start PowerCLI and run scripts written on the local computer with the `Set-ExecutionPolicy -ExecutionPolicy`

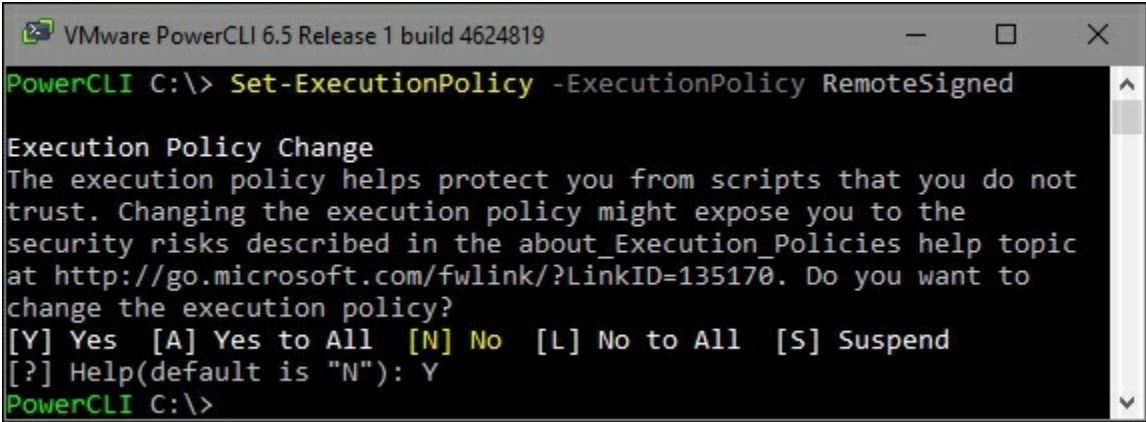
Note

You have to run the `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned` command from a PowerShell or PowerCLI session that you started using the **Run as Administrator** option, or you will get the following error message:

Set-ExecutionPolicy : Access to the registry key 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell' is denied.

If you are using both the 32-bit and the 64-bit versions of PowerCLI, you have to run this command in both versions.

In the following screenshot of the PowerCLI console, you will see the output of the `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned` command if you run this command in a PowerCLI session started with **Run as Administrator**.



You can get more information about execution policies by typing the following command:

```
PowerCLI C:\> Get-Help about_Execution_Policies
```

To get more information about signing your scripts, type the following command:

```
PowerCLI C:\> Get-Help about_signing
```

Note

If you get an error message saying **Get-Help could not find about_Execution_Policies in a help file**, you have to run the `Update-Help` cmdlet in a PowerShell, or PowerCLI session started with **Run as Administrator** first. The `Update-Help` cmdlet downloads the newest help files for Microsoft PowerShell modules and installs them on your computer. Because Microsoft updates the Microsoft PowerShell help files on a regular basis, it is recommended to run the `Update-Help` cmdlet on a regular basis also.

Creating a PowerShell profile

If you want certain PowerCLI commands to be executed every time you start a PowerCLI session, you can put these commands in a **PowerShell profile**. The commands in a PowerShell profile will be executed every time you start a new PowerCLI session. There are six PowerShell profiles, two specific for the PowerShell console, two specific for the PowerShell **Integrated Scripting Environment (ISE)**, and two used by both the PowerShell console and the PowerShell ISE. The PowerShell console and the PowerShell ISE have their own profiles for:

- All users, current host
- Current user, current host

The two profiles used by both the PowerShell console and the PowerShell ISE are:

- All users, all hosts
- Current user, all hosts

You can retrieve the locations for the different profiles of the PowerShell console by executing the following command in the PowerShell console. In this command, the `$PROFILE` variable is a standard PowerShell variable that returns an object containing the locations of the PowerShell profiles. This object is piped to the `Format-List -Force` command to display all of the properties of the `$PROFILE` object in a list:

```
PowerCLI C:\> $PROFILE | Format-List -Force

AllUsersAllHosts      :
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost   :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts   :
C:\Users\robert\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\robert\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                : 76
```

Note

Downloading the example code

Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look.

The code bundle for the book is also hosted on GitHub at:

https://github.com/rosbook/effective_robotics_programming_with_ros. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

As you can see in the output of the preceding command, the `$PROFILE` object has four properties `AllUsersAllHosts`, `AllUsersCurrentHost`, `CurrentUserAllHosts`, and

`CurrentUserCurrentHost` that contain the locations of the different profiles. To list the locations of the PowerShell profiles of the PowerShell ISE, you have to execute the preceding command in the PowerShell ISE. This gives the following output:

```
PS C:\> $PROFILE | Format-List -Force

AllUsersAllHosts      :
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost   :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1
CurrentUserAllHosts   :
C:\Users\robert\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\robert\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
Length                : 79
```

Note

You can start the PowerShell ISE from a Command Prompt by running `powershell_ise.exe`. You can start the PowerShell ISE from within a PowerShell console with the alias `ise`.

The default value for the `$PROFILE` variable is the value of the `$PROFILE.CurrentUserCurrentHost` property. So you can use `$PROFILE` instead of `$PROFILE.CurrentUserCurrentHost`.

You can determine if a specific profile exists by using the `Test-Path` cmdlet. The following command will test if the profile specified by `$PROFILE` exists:

```
PowerCLI C:\> Test-Path -Path $PROFILE
False
```

If a profile does not exist, as in the preceding example, you can create the profile using the `New-Item` cmdlet. If the directories in the path do not exist, by using the `-Force` parameter the `New-Item` cmdlet will create the directories. The following command will create the current user/current host profile and will also create the missing directories in the path:

```
PowerCLI C:\> New-Item -Path $PROFILE -ItemType file -Force

Directory: C:\Users\robert\Documents\WindowsPowerShell

Mode                LastWriteTime         Length Name
----                -
-a--              1/7/2017   2:01 PM             0
Microsoft.PowerShell_pro
                                file.ps1
```

After creating the PowerShell profile, you can edit the profile using the PowerShell ISE with the following command:

```
PowerCLI C:\> ise $PROFILE
```


If you put the commands from the preceding section, Modifying the PowerShell execution policy, the new colors of the messages will be used in all of your PowerCLI sessions.

Connecting and disconnecting servers

Before you can do useful things with PowerCLI, you have to connect to a vCenter Server or an ESXi server. And if you are finished, it is a good practice to disconnect your session. We will discuss how to do this in the following sections Connecting to a server, Connecting to multiple servers, Suppressing certificate warnings , and Disconnecting from a server.

Connecting to a server

If you are not connected to a vCenter or an ESXi server, you will get an error message if you try to run a PowerCLI cmdlet. Let's try to retrieve a list of all of your data centers using the following command:

```
PowerCLI C:\> Get-Datacenter
```

The output of the preceding command is as follows:

```
Get-Datacenter : 1/7/2017 1:37:17 PM      Get-Datacenter      You
are
    not currently connected to any servers. Please connect first
using a
    Connect cmdlet.
At line:1 char:1
+ Get-Datacenter
+ ~~~~~
+ CategoryInfo          : ResourceUnavailable: (:) [Get-
Datacenter],
ViServerConnectionException
+ FullyQualifiedErrorId : Core_BaseCmdlet_NotConnectedError,

VMware.VimAutomation.ViCore.Cmdlets.
Commands.GetDatacenter
```

You can see that this gives an error message. You first have to connect to a vCenter Server or an ESXi server using the `Connect-VIServer` cmdlet. If you have a vCenter Server, you only need to connect to the vCenter Server and not to the individual ESXi servers. It is possible to connect to multiple vCenter Servers or ESXi servers at once. The `Connect-VIServer` cmdlet has the following syntax. The syntax contains two parameter sets. The first parameter set is the default:

```
Connect-VIServer [-Server] <String[]> [-Port <Int32>] [-Protocol
    <String>] [-Credential <PSCredential>] [-User <String>] [-
Password
    <String>] [-Session <String>] [-NotDefault] [-SaveCredentials]
    [-AllLinked] [-Force] [<CommonParameters>]
```

In the `Default` parameter set, the `-Server` parameter is required. The second parameter set can be used to select a server from a list of recently connected servers:

```
Connect-VIServer -Menu [<CommonParameters>]
```

In the `Menu` parameter set, the `-Menu` parameter is required. You cannot combine parameters from the `Default` parameter set with the `Menu` parameter set. Let's first try to connect to a vCenter Server with the following command:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132
```

192.168.0.132 is the IP address of the vCenter Server in my home lab. Replace this IP address with the IP address or DNS name of your vCenter or ESXi server. The preceding command will pop up a window in which you have to specify server credentials to connect to your server if your Windows session credentials don't have rights on your server. Enter values for **User name** and **Password** and click on **OK**. If you specified valid credentials, you would get output similar to the following:

Name	Port	User
-----	----	----
192.168.0.132	443	root

You can also specify a username and password on the command line as follows:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132 -User admin  
-Password pass
```

You can also save the credentials in a variable with the following command:

```
PowerCLI C:\> $Credential = Get-Credential
```

The preceding command will pop up a window in which you can type the username and password.



You can now use the `$Credential` variable to connect to a server using the `-Credential` parameter, as follows:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132 -Credential  
$Credential
```

You can also use the PowerCLI credential store. This will be discussed in the Using the credential store section, later in this chapter.

The default protocol that the `Connect-VIServer` cmdlet uses is HTTPS. If you want to make a connection with the HTTP protocol, you can do that with the following command:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132 -Protocol HTTP
```

If you have multiple vCenter Servers in **Linked Mode**, you can use the `Connect-VIServer -AllLinked` parameter to connect all of these vCenter Servers at once, as follows:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132 -Credential  
$Credential -AllLinked
```

The `Connect-VIServer -Menu` command gives you a list of previously connected servers from which you can pick one, as shown in the following command line:

```
PowerCLI C:\> Connect-VIServer -Menu
Select a server from the list (by typing its number and pressing
Enter):
[1] 192.168.0.132
[2] 192.168.0.133
```

Type the number of the server you want to connect to.

Connecting to multiple servers

It is possible in PowerCLI to connect to multiple servers at once. You can do this by specifying more than one server, as follows:

```
PowerCLI C:\> Connect-VIServer -Server vCenter1,vCenter2
```

The first time you try to do this, you will get the following message:

```
Working with multiple default servers?
```

```
Select [Y] if you want to work with more than one default
servers. In this case, every time when you connect to a different
server using Connect-VIServer, the new server connection is stored
in
```

```
an array variable together with the previously connected servers.
When you run a cmdlet and the target servers cannot be determined
from the specified parameters, the cmdlet runs against all servers
stored in the array variable.
```

```
Select [N] if you want to work with a single default server. In
this case, when you run a cmdlet and the target servers cannot be
determined from the specified parameters, the cmdlet runs against
the
last connected server.
```

```
WARNING: WORKING WITH MULTIPLE DEFAULT SERVERS WILL BE
ENABLED BY
```

```
DEFAULT IN A FUTURE RELEASE. You can explicitly set your own
preference at any time by using the DefaultServerMode parameter of
Set-PowerCLIConfiguration.
```

```
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

Press **Enter** or type **Y** to work with multiple default servers.

As the message says, you can always connect to multiple servers, but your commands will only work against the last server you connected to unless you have enabled working with multiple servers.

You can see the current value of `DefaultVIServerMode` with the `Get-PowerCLIConfiguration` cmdlet:

```
PowerCLI C:\> Get-PowerCLIConfiguration
```

Scope	ProxyPolicy	DefaultVIServerMode
InvalidCertificateAction		

--		
Session	UseSystemProxy	Multiple
User		Unset
AllUsers		Multiple

If you want to change `DefaultVIServerMode` from single to multiple, you can do that with the `Set-PowerCLIConfiguration` cmdlet. This cmdlet has the following syntax:

```
Set-PowerCLIConfiguration [-ProxyPolicy <ProxyPolicy>]
[-DefaultVIServerMode <DefaultVIServerMode>] [-
InvalidCertificateAction
<BadCertificateAction>] [-ParticipateInCeip <Boolean>]
[-CEIPDataTransferProxyPolicy <ProxyPolicy>] [-
DisplayDeprecationWarnings <Boolean>] [-
WebOperationTimeoutSeconds
<Int32>] [-VMConsoleWindowBrowser <String>] [-Scope
<ConfigurationScope>] [-WhatIf] [-Confirm] [<CommonParameters>]
```

You can change `DefaultVIServerMode` from single to multiple with the following command:

```
PowerCLI C:\> Set-PowerCLIConfiguration -DefaultVIServerMode
Multiple
-Scope User
```

All of the servers that you are currently connected to are stored in the variable `$global:DefaultVIServers`. If `DefaultVIServerMode` is set to `multiple`, your PowerCLI cmdlets will run against all servers stored in the `$global:DefaultVIServers` variable. The last server you are connected to is stored in the variable `$global:DefaultVIServer`. If `DefaultVIServerMode` is set to `single`, your PowerCLI cmdlets will only run against the server stored in the `$global:DefaultVIServer` variable.

Suppressing certificate warnings

If your vCenter Server does not have valid server certificates, the `Connect-VIserver` cmdlet will display some warning messages, as shown in the following screenshot:

```
VMware PowerCLI 6.5 Release 1 build 4624819
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132
WARNING: There were one or more problems with the server certificate
for the server 192.168.0.132:443:

* The X509 chain could not be built up to the root certificate.
* The certificate's CN name does not match the passed value.

Certificate: [Subject]
             C=US, CN=vcenter.blackmilktea.com

[Issuer]
             OU=VMware Engineering, O=vcenter.blackmilktea.com, S=California,
             C=US, DC=local, DC=vsphere, CN=CA

[Serial Number]
             00CBA6FC3EE198B182

[Not Before]
             1/7/2017 11:57:31 AM

[Not After]
             1/2/2027 11:57:31 AM

[Thumbprint]
             ACE668E4BB0EAD332EE21B96533415E1D431F1CC

The server certificate is not valid.

WARNING: THE DEFAULT BEHAVIOR UPON INVALID SERVER CERTIFICATE WILL
CHANGE IN A FUTURE RELEASE. To ensure scripts are not affected by
the change, use Set-PowerCLIConfiguration to set a value for the
InvalidCertificateAction option.

Name                                Port  User
----                                -
192.168.0.132                      443   VSPHERE.LOCAL\Administrator
```

It is a good practice to supply your vCenter Server and ESXi servers with certificates signed by a **certificate authority (CA)**. You can find information on how to do this in the VMware Knowledge Base article, Replacing default certificates with CA signed SSL certificates in vSphere 6.x (2111219) at <http://kb.vmware.com/kb/2111219>. If you don't have valid certificates, you can suppress the warning messages using the `Set-PowerCLIConfiguration` cmdlet with the following command:

```
PowerCLI C:\> Set-PowerCLIConfiguration -InvalidCertificateAction
Ignore
```

The preceding command will modify `InvalidCertificateAction` in the `AllUsers` scope. You have to run this command using the **Run as Administrator** PowerCLI session. Otherwise, you will get the following error message:

```

Set-PowerCLIConfiguration : Only administrators can change settings
for
    all users.
At line:1 char:1
+ Set-PowerCLIConfiguration -InvalidCertificateAction Ignore
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:)
                        [Set-PowerCLIConfiguration],
                        InvalidArgument
+ FullyQualifiedErrorId : VMware.VimAutomation.ViCore.Types.V1.
                        ErrorHandling.InvalidArgument,
VMware.

VmAutomation.ViCore.Cmdlets.Commands.
                        SetVIToolkitConfiguration

```

Disconnecting from a server

To disconnect from a vSphere server, you have to use the `Disconnect-VIServer` cmdlet. The `Disconnect-VIServer` cmdlet has the following syntax:

```

Disconnect-VIServer [[-Server] <VIServer[]>] [-Force]
[-WhatIf] [-Confirm] [<CommonParameters>]

```

To disconnect all of your server connections, type the following command:

```

PowerCLI C:\> Disconnect-VIServer -Server * -Force

```

The output of the preceding command is as follows:

```

Confirm
Are you sure you want to perform this action?
Performing operation "Disconnect VIServer" on Target "User:
VSPHERE.LOCAL\Administrator, Server: 192.168.0.132, Port: 443".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend
[?] Help (default is "Y"):

```

Type **Y** or **Enter** to disconnect from the server.

If you don't want to be prompted with **Are you sure you want to perform this action?**, you can use the `-Confirm:$false` option as follows:

```

PowerCLI C:\> Disconnect-VIServer -Server * -Force
               -Confirm:$false

```

It may be that you want to disconnect only one session and not all. In that case, specify the server name or IP address of the server you want to disconnect. The following command only disconnects the latest session from server `192.168.0.132`:

```

PowerCLI C:\> Disconnect-VIServer -Server 192.168.0.132

```

Disconnecting one or more sessions will also change the value of the `$global:DefaultVIServers` and `$global:DefaultVIServer` variables.

Retrieving the PowerCLI configuration

To see the current setting of `InvalidCertificateAction`, you can use the `Get-PowerCLIConfiguration` cmdlet. The syntax of this cmdlet is as follows:

```
Get-PowerCLIConfiguration [-Scope <ConfigurationScope>]
[<CommonParameters>]
```

The following example will retrieve the PowerCLI configuration and shows the `InvalidCertificateAction` value for all scopes:

```
PowerCLI C:\> Get-PowerCLIConfiguration |
>> Select-Object -Property Scope, InvalidCertificateAction,
    DisplayDeprecationWarnings |
>> Format-Table -AutoSize
>>
```

Scope	InvalidCertificateAction	DisplayDeprecationWarnings
Session	Unset	True
User		
AllUsers		

As you can see in the output, there are three different scopes for which you can modify the PowerCLI configuration: `Session`, `User`, and `AllUsers`. The `Set-PowerCLIConfiguration` cmdlet will modify the `AllUser` scope if you don't specify a scope.

The `DisplayDeprecationWarnings` property shown in the preceding output will be discussed in the section, `Suppressing deprecated warnings`, later in this chapter.

Using the credential store

If you are logged in to your computer with a domain account, you can use your Windows session credentials to connect to a vCenter or ESXi server. If you are not logged in to your computer with a domain account or your domain account has no rights in vSphere, you have to supply account information every time you connect to a vCenter or ESXi server.

To prevent yourself from having to do this, you can store credentials in the credential store. These stored credentials will be used as default if you connect to a server that is stored in the credential store. You can use the `-SaveCredentials` parameter of the `Connect-VIServer` cmdlet to indicate that you want to save the specified credentials in the local credential store, as follows:

```
PowerCLI C:\> Connect-VIServer -Server 192.168.0.132 -User admin  
-Password pass -SaveCredentials
```

You can also create a new entry in the credential store with the `New-VICredentialStoreItem` cmdlet:

```
PowerCLI C:\> New-VICredentialStoreItem -Host 192.168.0.132  
-User Admin -Password pass
```

You can not only store credentials for vCenter Servers but also for ESXi servers, using the following command:

```
PowerCLI C:\> New-VICredentialStoreItem -Host ESX1 -User root  
-Password VMware1!
```

To get a listing of all of your stored credentials, type the following command:

```
PowerCLI C:\> Get-VICredentialStoreItem
```

And to remove a stored credential you can use the following command:

```
PowerCLI C:\> Remove-VICredentialStoreItem -Host ESX1 -User root
```

The stored credentials are stored in a file on your computer. The default credential store file location is: `%APPDATA%\VMware\credstore\vicredentials.xml`. But it is also possible to create other credential store files. You can see the contents of the default credential store file with the following command:

```
PowerCLI C:\> Get-Content -Path $env:APPDATA\VMware\credstore  
\vicredentials.xml
```

The passwords stored in a credential store file are encrypted. But you can easily retrieve the stored passwords with the following command:

```
PowerCLI C:\> Get-VICredentialStoreItem |  
>> Select-Object -Property Host,User,Password
```

Note

The passwords in the stored credentials are encrypted. Only the user who created the item can decrypt the password.

Retrieving a list of all of your virtual machines

Now that we know how to connect to a server, let's do something useful with PowerCLI. Most of the people who begin using PowerCLI create reports, so create a list of all of your virtual machines as your first report. You have to use the `Get-VM` cmdlet to retrieve a list of your virtual machines. The syntax of the `Get-VM` cmdlet is as follows. The first parameter set is the default:

```
Get-VM [[-Name] <String[]>] [-Server <VIServer[]>]
      [-Datastore <StorageResource[]>] [-Location <VIContainer[]>]
      [-Tag <Tag[]>] [-NoRecursion] [<CommonParameters>]
```

The second parameter set is for retrieving virtual machines connected to specific virtual switches:

```
Get-VM [[-Name] <String[]>] [-Server <VIServer[]>] [-VirtualSwitch
      <VirtualSwitchBase[]>] [-Tag <Tag[]>] [<CommonParameters>]
```

The third parameter set is for retrieving virtual machines by ID:

```
Get-VM [-Server <VIServer[]>] -Id <String[]> [<CommonParameters>]
```

The `-Id` parameter is required. The fourth parameter set is for retrieving virtual machines by related object:

```
Get-VM -RelatedObject <VmRelatedObjectBase[]> [<CommonParameters>]
```

The `-RelatedObject` parameter is required. You can use these four parameter sets to filter the virtual machines based on name, server, datastore, location, distributed switch, ID, or related object.

Create your first report with the following command:

```
PowerCLI C:\> Get-VM
```

This will create a list of all of your virtual machines. You will see the name, power state, the number of CPU's, and the amount of memory in GB for each virtual machine, as shown in the following command-line output:

Name	PowerState	NumCPUs	MemoryGB
----	-----	-----	-----
Dc1	PoweredOn	2	4.000
VM1	PoweredOn	1	0.250
DNS1	PoweredOn	2	8.000

The `Name`, `PowerState`, `NumCPU`, and `MemoryGB` properties are the properties that you will see by default if you use the `Get-VM` cmdlet. However, the virtual machine object in PowerCLI has a lot of other properties that are not shown by default. You can see them all by piping the output of the `Get-VM` cmdlet to the `Format-List` cmdlet using the pipe character `|`. The `Format-List` cmdlet displays object properties and their values in a list format, as shown in the following command-line output:

```
PowerCLI C:\> Get-VM -Name DC1 | Format-List -Property *
```

```

      Name                               : DC1
PowerState                               : PoweredOff
Notes                                    :
Guest                                    : DC1:
NumCPU                                   : 1
CoresPerSocket                           : 1
MemoryMB                                 : 4096
MemoryGB                                 : 4
VMHostId                                 : HostSystem-host-10
VMHost                                   : 192.168.0.133
VApp                                     :
FolderId                                 : Folder-group-v9
Folder                                   : Discovered virtual machine
ResourcePoolId                           : ResourcePool-resgroup-8
ResourcePool                             : Resources
HARestartPriority                         : ClusterRestartPriority
HAIsolationResponse                      : AsSpecifiedByCluster
DrsAutomationLevel                       : AsSpecifiedByCluster
VMSwapfilePolicy                         : Inherit
VMResourceConfiguration                  : CpuShares:Normal/1000
                                          MemShares:Normal/40960
Version                                  : v13
PersistentId                             : 50399fa1-6d65-a26f-1fd2-b635d0e8610f
GuestId                                  : windows9Server64Guest
UsedSpaceGB                              : 30.000001807697117328643798828
ProvisionedSpaceGB                       : 34.175447797402739524841308594
DatastoreIdList                          : {Datastore-datastore-11}
ExtensionData                            : VMware.Vim.VirtualMachine
CustomFields                             : {}
Id                                         : VirtualMachine-vm-46
Uid                                       :
/VIServer=vsphere.local/administrator@192.
168.0.132:443/VirtualMachine=VirtualMachin
e-vm-46/
Client                                   :
VMware.VimAutomation.ViCore.Impl.V1.VimCli
Ent
```

You can select specific properties with the `Select-Object` cmdlet. Say you want to make a report that shows the `Name`, `Notes`, `VMHost`, and `Guest` properties for all your virtual machines. You can do that with the following command:

```
PowerCLI C:\> Get-VM | Select-Object -Property
Name,Notes,VMHost,Guest
```

The output of the preceding command is as follows:

Name	Notes	VMHost	Guest
----	-----	-----	-----
DC1		192.168.0.133	DC1:
VM1		192.168.0.134	VM1:
DNS1	DNS Server	192.168.0.134	DNS1:

In PowerShell, parameters can be positional. This means that you can omit the parameter name if you put the parameter values in the right order. In the preceding example, the `-Property` parameter of the `Select-Object` cmdlet can be omitted. So the preceding command can also be written as:

```
PowerCLI C:\> Get-VM | Select-Object Name,Notes,VMHost,Guest
```

In the examples in this book, I will always use the parameter names.

Suppressing deprecated warnings

You will probably have also seen the following warning messages:

```
WARNING: The 'Description' property of VirtualMachine type is
         deprecated. Use the 'Notes' property instead.
WARNING: The 'HardDisks' property of VirtualMachine type is
         deprecated. Use 'Get-HardDisk' cmdlet instead.
WARNING: The 'NetworkAdapters' property of VirtualMachine type is
         deprecated. Use 'Get-NetworkAdapter' cmdlet instead.
WARNING: The 'UsbDevices' property of VirtualMachine type is
         deprecated. Use 'Get-UsbDevice' cmdlet instead.
WARNING: The 'CDDrives' property of VirtualMachine type is
         deprecated. Use 'Get-CDDrive' cmdlet instead.
WARNING: The 'FloppyDrives' property of VirtualMachine type is
         deprecated. Use 'Get-FloppyDrive' cmdlet instead.
WARNING: The 'Host' property of VirtualMachine type is deprecated.
         Use the 'VMHost' property instead.
WARNING: The 'HostId' property of VirtualMachine type is
         deprecated.
         Use the 'VMHostId' property instead.
WARNING: PowerCLI scripts should not use the 'Client' property. The
         property will be removed in a future release.
```

These warning messages show the properties that should not be used in your scripts because they are deprecated and might be removed in a future PowerCLI release. Personally, I like these warnings because they remind me of the properties that I should not use anymore. But if you don't like these warnings, you can stop them from appearing with the following command:

```
PowerCLI C:\> Set-PowerCLIConfiguration -DisplayDeprecationWarnings
           $false -Scope User
```

Using wildcard characters

You can also use **wildcard characters** to select specific virtual machines. To display only the virtual machines that have names that start with an `A` or `a`, type the following command:

```
PowerCLI C:\> Get-VM -Name A*
```

Parameter values are not case sensitive. The asterisk (`*`) is a wildcard character that matches zero or more characters, starting at the specified position. Another wildcard

character is the question mark (?), which matches any character at the specified position. To get all virtual machines with a three-letter name that ends with e, use the following command:

```
PowerCLI C:\> Get-VM -Name ??e
```

You can also specify some specific characters, as shown in the following command:

```
PowerCLI C:\> Get-VM -Name [bc]*
```

The preceding command displays all of the virtual machines that have names starting with b or c. You can also specify a range of characters, as shown in the following command:

```
PowerCLI C:\> Get-VM -Name *[0-4]
```

The preceding command lists all of the virtual machines that have names ending with 0, 1, 2, 3, or 4.

Filtering objects

If you want to filter properties that don't have their own Get-VM parameter, you can pipe the output of the Get-VM cmdlet to the Where-Object cmdlet. Using the Where-Object cmdlet, you can set the filter on any property. Let's display a list of all of your virtual machines that have more than one virtual CPU using the following command:

```
PowerCLI C:\> Get-VM | Where-Object {$_.NumCPU -gt 1}
```

In this example, the Where-Object cmdlet has a PowerShell scriptblock as a parameter. A **scriptblock** is a PowerShell script surrounded by braces. In this scriptblock, you see \$_. When using commands in the pipeline, \$_ represents the current object. In the preceding example, \$_ represents the virtual machine object that is passed through the pipeline. \$_.NumCPU is the NumCPU property of the current virtual machine in the pipeline. -gt means greater than, so the preceding command shows all virtual machines' objects where the NumCPU property has a value greater than 1. PowerShell V3 introduced a new, easier syntax for the Where-Object cmdlet. You don't have to use a scriptblock anymore. You can now use the following command:

```
PowerCLI C:\> Get-VM | Where-Object NumCPU -gt 1
```

Isn't the preceding command much more like simple English?

Note

In the rest of this book, the PowerShell V2 syntax will be used by default because the V2 syntax will also work in PowerShell V3 and higher versions of PowerShell. If PowerShell V3 syntax is used anywhere, it will be specifically mentioned.

Using comparison operators

In the preceding section, Filtering objects, you saw an example of the -gt comparison operator. In the following table, we will show you all of the PowerShell comparison operators:

Operator	Description
-eq, -ceq, and -ieq	Equal to.

-ne, -cne, and -ine	Not equal to.
-gt, -cgt, and -igt	Greater than.
-ge, -cge, and -ige	Greater than or equal to.
-lt, -clt, and -ilt	Less than.
-le, -cle, and -ile	Less than or equal to.
-Like	Match using the wildcard character (*).
-NotLike	Does not match using the wildcard character (*).
-Match	Matches a string using regular expressions.
-NotMatch	Does not match a string. Uses regular expressions.
-Contains	Tells whether a collection of reference values includes a single test value.
-NotContains	Tells whether a collection of reference values does not include a single test value.
-In	Tells whether a test value appears in a collection of reference values.
-NotIn	Tells whether a test value does not appear in a collection of reference values.

In the preceding table, you see three different operators for some functions. So what is the difference? The `c` variant is case sensitive. The two-letter variant and the `i` variant are case-insensitive. The `i` variant is made to make it clear that you want to use the case insensitive operator.

Using aliases

The `Where-Object` cmdlet has two aliases: `?` and `where`. Therefore, both the following commands will display a list of all your virtual machines that have more than one virtual CPU:

```
PowerCLI C:\> Get-VM | ? {$_.NumCPU -gt 1}
```

```
PowerCLI C:\> Get-VM | Where NumCPU -gt 1
```

Note

Using aliases will save you from the trouble of typing in the PowerCLI console. However, it is good practice to use the full cmdlet names when you write a script. This will make the script much more readable and easier to understand.

To see a list of all of the aliases that are defined for cmdlets, type the following command:

```
PowerCLI C:\> Get-Alias
```

You can create aliases using the `New-Alias` cmdlet. For example, to create an alias `childs` for the `Get-ChildItem` cmdlet, you can use the following command:

```
PowerCLI C:\> New-Alias -Name childs -Value Get-ChildItem
```

Retrieving a list of all of your hosts

Similar to the `Get-VM` cmdlet, which retrieves your virtual machines, is the `Get-VMHost` cmdlet, which displays your hosts. The `Get-VMHost` cmdlet has the following syntax. The first parameter set is the default:

```
Get-VMHost [[-Name] <String[]>] [-NoRecursion] [-Datastore
    <StorageResource[]>] [-State <VMHostState[]>] [-Location
    <Vicontainer[]>]
    [-Tag <Tag[]>] [-Server <VIMServer[]>] [<CommonParameters>]
```

The second parameter set is for retrieving hosts connected to specific distributed virtual switches:

```
Get-VMHost [[-Name] <String[]>] [-DistributedSwitch
    <DistributedSwitch[]>] [-Tag <Tag[]>] [-Server <VIMServer[]>]
    [<CommonParameters>]
```

The third parameter set is for retrieving hosts by virtual machine or resource pool:

```
Get-VMHost [[-Name] <String[]>] [-NoRecursion] [-VM
    <VirtualMachine[]>]
    [-ResourcePool <ResourcePool[]>] [-Datastore
    <StorageResource[]>]
    [-Location <Vicontainer[]>] [-Tag<Tag[]>] [-Server
    <VIMServer[]>]
    [<CommonParameters>]
```

The fourth parameter set is for retrieving hosts by ID:

```
Get-VMHost [-Server <VIMServer[]>] -Id <String[]>
    [<CommonParameters>]
```

The `-Id` parameter is required. The fifth parameter set is for retrieving hosts by related object:

```
Get-VMHost [-RelatedObject] <VMHostRelatedObjectBase[]>
    [<CommonParameters>]
```

The `-RelatedObject` parameter is required.

Don't mix parameters from different sets or you will get an error as follows:

```
PowerCLI C:\> Get-VMHost -Id HostSystem-host-22 -Name 192.168.0.133
Get-VMHost : Parameter set cannot be resolved using the specified
named
    parameters.
At line:1 char:1
+ Get-VMHost -Id HostSystem-host-22 -Name 192.168.0.133
+ ~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-VMHost],
    ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,
    VMware.VimAutomation

    .ViCore.Cmdlets.Commands.GetVMHost
```

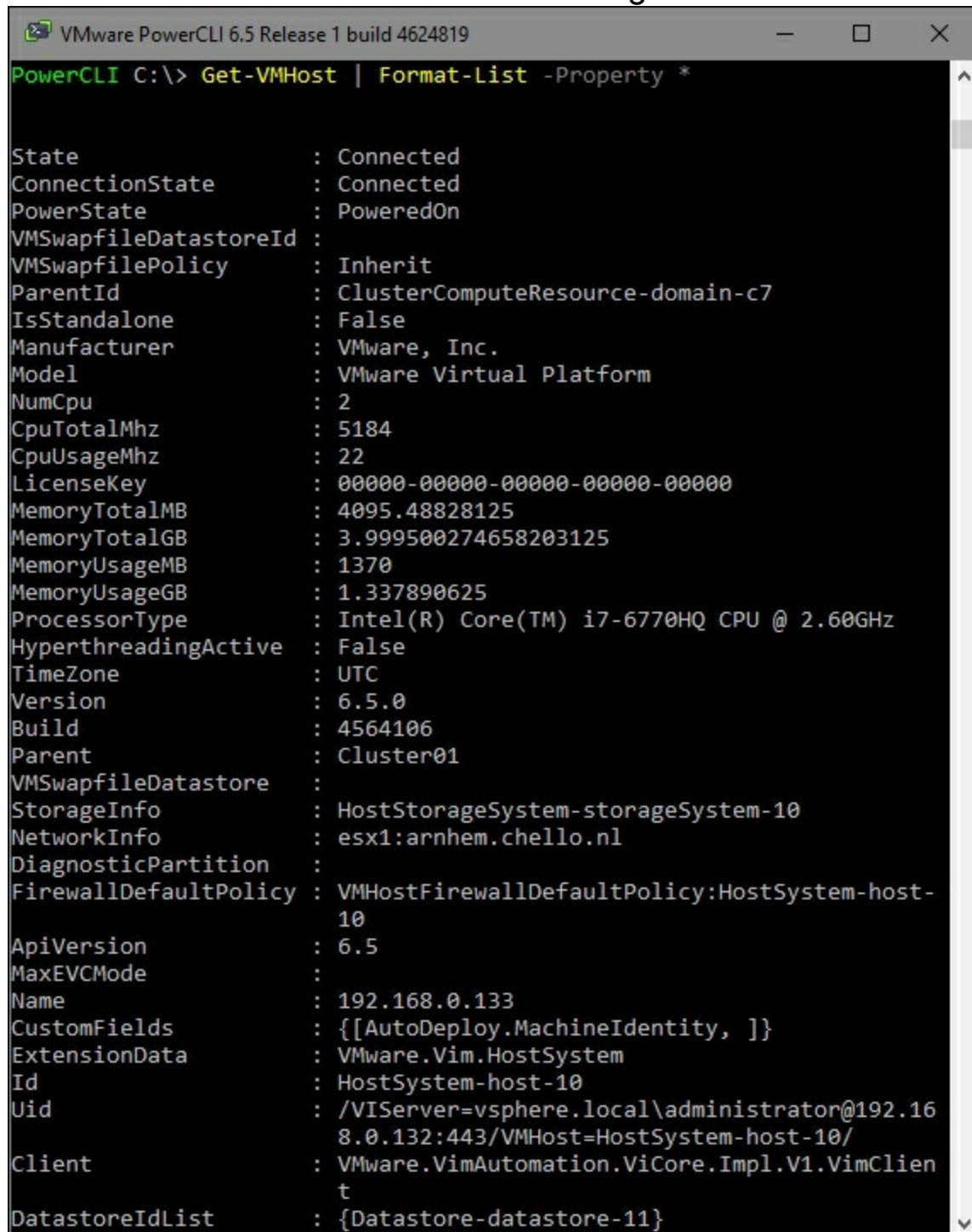
To get a list of all of your hosts, type the following command:

```
PowerCLI C:\> Get-VMHost
```


By default, only the `Name`, `ConnectionState`, `PowerState`, `NumCPU`, `CpuUsageMhz`, `CpuTotalMhz`, `MemoryUsageGB`, `MemoryTotalGB`, and `Version` properties are shown. To get a list of all of the properties, type the following command:

```
PowerCLI C:\> Get-VMHost | Format-List -Property *
```

The output of this command can be seen in the following screenshot:



```
VMware PowerCLI 6.5 Release 1 build 4624819
PowerCLI C:\> Get-VMHost | Format-List -Property *

State                : Connected
ConnectionState      : Connected
PowerState           : PoweredOn
VMSwapfileDatastoreId : 
VMSwapfilePolicy      : Inherit
ParentId             : ClusterComputeResource-domain-c7
IsStandalone         : False
Manufacturer         : VMware, Inc.
Model                : VMware Virtual Platform
NumCpu               : 2
CpuTotalMhz          : 5184
CpuUsageMhz          : 22
LicenseKey           : 00000-00000-00000-00000-00000
MemoryTotalMB        : 4095.48828125
MemoryTotalGB        : 3.999500274658203125
MemoryUsageMB        : 1370
MemoryUsageGB        : 1.337890625
ProcessorType        : Intel(R) Core(TM) i7-6770HQ CPU @ 2.60GHz
HyperthreadingActive : False
TimeZone             : UTC
Version              : 6.5.0
Build                : 4564106
Parent               : Cluster01
VMSwapfileDatastore  : 
StorageInfo          : HostStorageSystem-storageSystem-10
NetworkInfo          : esx1:arnhem.chello.nl
DiagnosticPartition   : 
FirewallDefaultPolicy : VMHostFirewallDefaultPolicy:HostSystem-host-10
ApiVersion           : 6.5
MaxEVCMode           : 
Name                 : 192.168.0.133
CustomFields          : {[AutoDeploy.MachineIdentity, ]}
ExtensionData         : VMware.Vim.HostSystem
Id                   : HostSystem-host-10
Uid                  : /VIServer=vsphere.local\administrator@192.16
                        8.0.132:443/VMHost=HostSystem-host-10/
Client                : VMware.VimAutomation.ViCore.Impl.V1.VimClient
DatastoreIdList       : {Datastore-datastore-11}
```

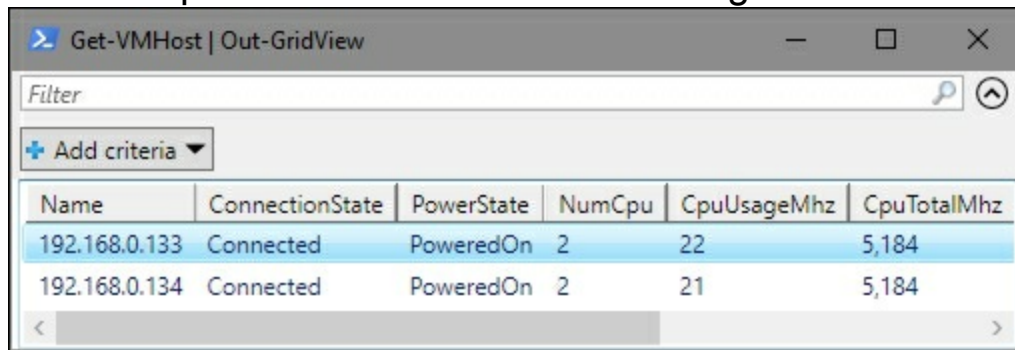
You can use the `Get-VMHost` parameters or the `Where-Object` cmdlet to filter the hosts you want to display, as we did with the `Get-VM` cmdlet.

Displaying the output in a grid view

Instead of displaying the output of your PowerCLI commands in the PowerCLI console, you can also display the output in a grid view. A grid view is a popup that looks like a spreadsheet with rows and columns. To display the output of the `Get-VMHost` cmdlet in a grid view, type the following command:

```
PowerCLI C:\> Get-VMHost | Out-GridView
```

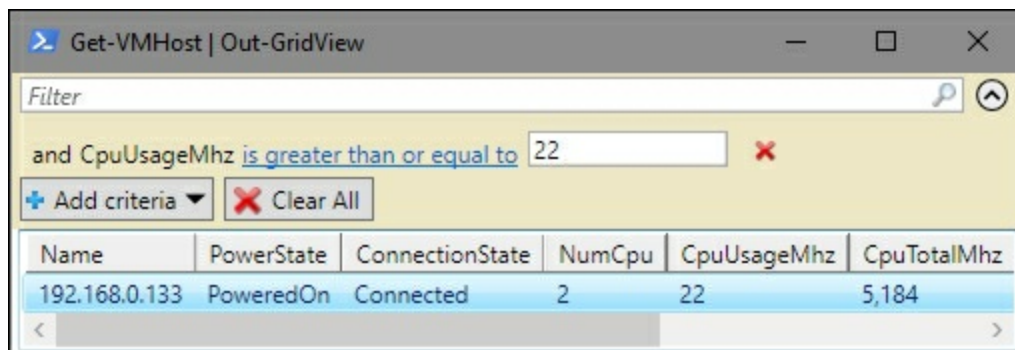
The preceding command opens the window of the following screenshot:



The screenshot shows a window titled 'Get-VMHost | Out-GridView'. It has a search bar at the top with the text 'Filter'. Below the search bar is a button labeled '+ Add criteria'. The main area contains a table with the following data:

Name	ConnectionState	PowerState	NumCpu	CpuUsageMhz	CpuTotalMhz
192.168.0.133	Connected	PoweredOn	2	22	5,184
192.168.0.134	Connected	PoweredOn	2	21	5,184

You can create filters to display only certain rows, and you can sort columns by clicking on the column header. You can also reorder columns by dragging and dropping them. In the following screenshot, we created a filter to show only the hosts with a **CpuUsageMhz** value greater than or equal to **22**. We also changed the order of the **ConnectionState** and **PowerState** columns.



The screenshot shows the same window as before, but with a filter applied. The filter bar now contains the text 'and CpuUsageMhz is greater than or equal to 22'. Below the filter bar is a button labeled '+ Add criteria' and a button labeled 'Clear All'. The table below shows only one row, which is the first row from the previous screenshot:

Name	PowerState	ConnectionState	NumCpu	CpuUsageMhz	CpuTotalMhz
192.168.0.133	PoweredOn	Connected	2	22	5,184

Isn't that cool?

Summary

In this chapter, we looked at downloading and installing PowerCLI, participating in the VMware CEIP, and modifying the PowerShell execution policy to be able to start PowerCLI. You learned to create a PowerShell profile containing commands that run every time you start PowerShell or PowerCLI. We showed you how to connect to and disconnect from a server and introduced the credential store to save you from having to specify credentials when you connect to a server. You also learned how to get a list of your virtual machines or hosts and how to stop deprecated warnings. You learned to filter objects by using the PowerShell comparison operators and found out about aliases for cmdlets. Finally, we concluded the chapter with grid views. In the next chapter, we will introduce some basic PowerCLI concepts.

Chapter 2. Learning Basic PowerCLI Concepts

While learning something new, you always have to learn the basics first. In this chapter, you will learn some basic PowerShell and PowerCLI concepts. Knowing these concepts will make it easier for you to learn the advanced topics. We will cover the following topics in this chapter:

- Using the `Get-Command`, `Get-Help`, and `Get-Member` cmdlets
- Using providers and PSdrives
- Using arrays and hash tables
- Creating calculated properties
- Using raw API objects with `ExtensionData` or `Get-View`
- Extending PowerCLI objects with the `New-VIProperty` cmdlet
- Working with vSphere folders

Using the Get-Command, Get-Help, and Get-Member cmdlets

There are some PowerShell cmdlets that everyone should know. Knowing these cmdlets will help you discover other cmdlets, their functions, parameters, and returned objects.

Using Get-Command

The first cmdlet that you should know is `Get-Command`. This cmdlet returns all the commands that are installed on your computer. The `Get-Command` cmdlet has the following syntax. The first parameter set is named `CmdletSet`:

```
Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-All] [-CommandType {Alias | Function | Filter | Cmdlet | ExternalScript | Application | Script | Workflow | Configuration | All}] [-FullyQualifiedModule <ModuleSpecification[]>] [-ListImported] [-Module <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-ShowCommandInfo] [-Syntax] [-TotalCount <Int32>] [<CommonParameters>]
```

The second parameter set is named `AllCommandSet`:

```
Get-Command [[-ArgumentList] <Object[]>] [-All] [-FullyQualifiedModule <ModuleSpecification[]>] [-ListImported] [-Module <String[]>] [-Noun <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-ShowCommandInfo] [-Syntax] [-TotalCount <Int32>] [-Verb <String[]>] [<CommonParameters>]
```

If you type the following command, you will get a list of commands installed on your computer, including cmdlets, aliases, functions, workflows, filters, scripts, and applications:

```
PowerCLI C:\> Get-Command
```

You can also specify the name of a specific cmdlet to get information about that cmdlet, as shown in the following command:

```
PowerCLI C:\> Get-Command -Name Get-VM | Format-Table -AutoSize
```

The preceding command returns the following information about the `Get-VM` cmdlet:

CommandType	Name	Version	Source
Cmdlet	Get-VM	6.5.0.2604913	VMware.VimAutomation.Core

You see that the command returns the command type and the name of the module that contains the `Get-VM` cmdlet. `CommandType`, `Name`, `Version`, and `Source` are the properties that the `Get-Command` cmdlet returns by default. You will get more properties if you pipe the output to the `Format-List` cmdlet. The following screenshot will show you the output of the `Get-Command -Name Get-VM | Format-List *` command:

PowerCLI C:\> **Get-Command -Name Get-VM | Format-List ***

```

HelpUri           : http://www.vmware.com/support/developer/PowerCLI/PowerCLI65R1/html/Get-VM.html
DLL               : C:\Program Files (x86)\VMware\Infrastructure\PowerCLI\Modules\VMware.VimAutomation.Core\VMware.VimAutomation.ViCore.Cmdlets.dll
Verb              : Get
Noun              : VM
HelpFile          : C:\Program Files (x86)\VMware\Infrastructure\PowerCLI\Modules\VMware.VimAutomation.Core\VMware.VimAutomation.ViCore.Cmdlets.dll-Help.xml
PSSnapIn          :
Version           : 6.5.0.2604913
ImplementingType   : VMware.VimAutomation.ViCore.Cmdlets.Commands.GetVM
Definition        :
                    Get-VM [[-Name] <string[]>] [-Server <VIServer[]>] [-Datastore <StorageResource[]>] [-Location <VIContainer[]>] [-Tag <Tag[]>] [-NoRecursion] [<CommonParameters>]

                    Get-VM [[-Name] <string[]>] [-Server <VIServer[]>] [-VirtualSwitch <VirtualSwitchBase[]>] [-Tag <Tag[]>] [<CommonParameters>]

                    Get-VM [-Server <VIServer[]>] [-Id <string[]>] [<CommonParameters>]

                    Get-VM -RelatedObject <VmRelatedObjectBase[]> [<CommonParameters>]

DefaultParameterSet : Default
OutputType           : {VMware.VimAutomation.ViCore.Types.V1.Inventory.VirtualMachine}
Options              : ReadOnly
Name                  : Get-VM
CommandType          : Cmdlet
Source                : VMware.VimAutomation.Core
Visibility            : Public
ModuleName            : VMware.VimAutomation.Core
Module                : VMware.VimAutomation.Core
RemotingCapability    : PowerShell
Parameters            : {[Name, System.Management.Automation.ParameterMetadata], [Server, System.Management.Automation.ParameterMetadata], [Datastore, System.Management.Automation.ParameterMetadata], [VirtualSwitch, System.Management.Automation.ParameterMetadata]...}
ParameterSets        : {[[[-Name] <string[]>] [-Server <VIServer[]>] [-Datastore <StorageResource[]>] [-Location <VIContainer[]>] [-Tag <Tag[]>] [-NoRecursion] [<CommonParameters>], [[[-Name] <string[]>] [-Server <VIServer[]>] [-VirtualSwitch <VirtualSwitchBase[]>] [-Tag <Tag[]>] [<CommonParameters>], [-Server <VIServer[]>] [-Id <string[]>] [<CommonParameters>], -RelatedObject <VmRelatedObjectBase[]> [<CommonParameters>]]}

```

You can use the `Get-Command` cmdlet to search for cmdlets. For example, if necessary, search for the cmdlets that are used for vSphere hosts. Type the following command:

```
PowerCLI C:\> Get-Command -Name *VMHost*
```

If you are searching for the cmdlets to work with networks, use the following command:

```
PowerCLI C:\> Get-Command -Name *network*
```

Using Get-VICommand

PowerCLI has a `Get-VICommand` cmdlet that is similar to the `Get-Command` cmdlet. The `Get-VICommand` cmdlet is a function that creates a filter on the `Get-Command` output, and it returns only PowerCLI commands. Type the following command to list all the PowerCLI commands:

```
PowerCLI C:\> Get-VICommand
```

The `Get-VICommand` cmdlet has only one parameter `-Name`. So, you can also type, for example, the following command to get information only about the `Get-VM` cmdlet:

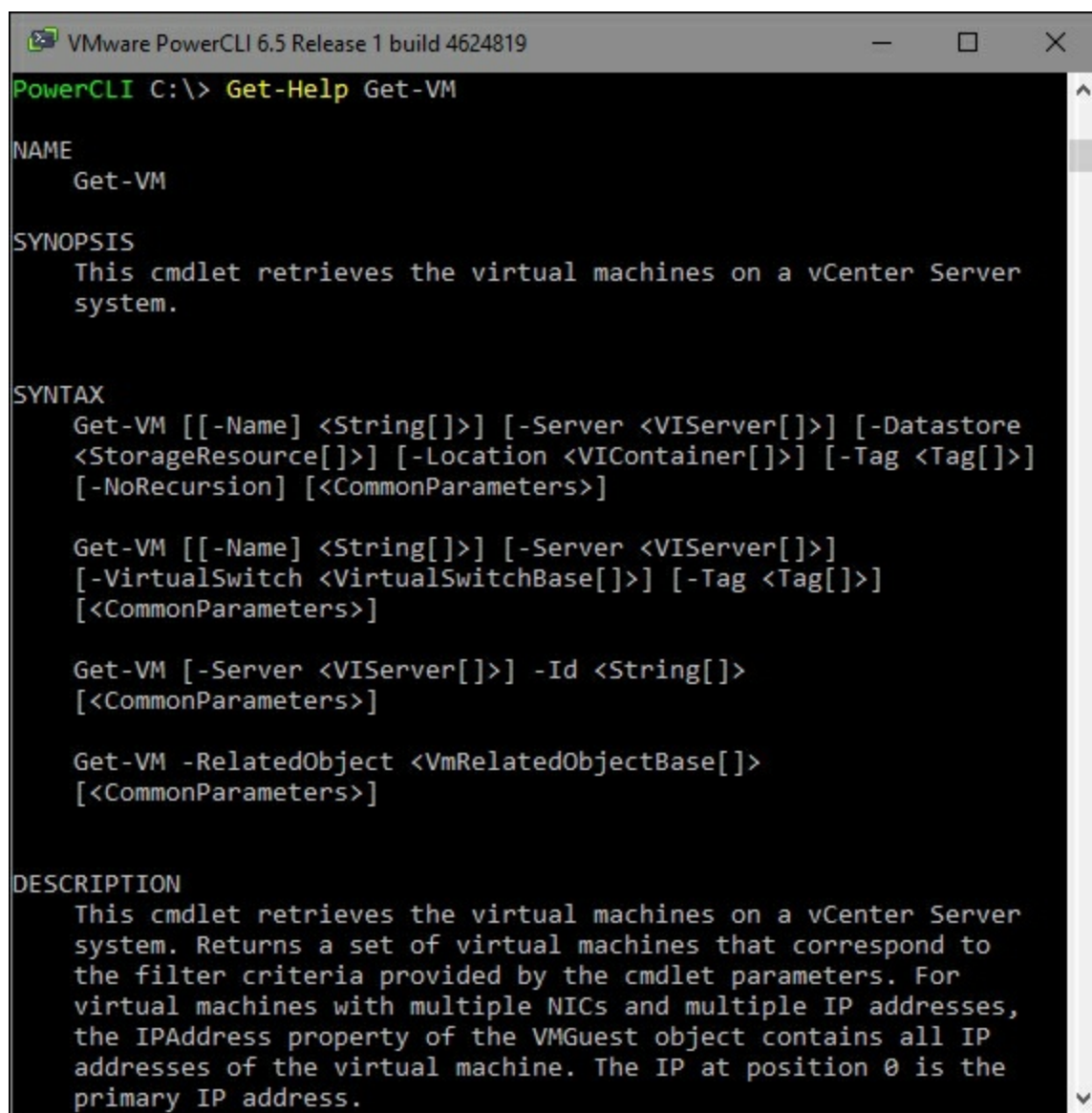
```
PowerCLI C:\> Get-VICommand -Name Get-VM
```

Using Get-Help

To discover more information about cmdlets, you can use the `Get-Help` cmdlet. For example:

```
PowerCLI C:\> Get-Help Get-VM
```

This will display the following information about the `Get-VM` cmdlet:



```
PowerCLI C:\> Get-Help Get-VM

NAME
    Get-VM

SYNOPSIS
    This cmdlet retrieves the virtual machines on a vCenter Server
    system.

SYNTAX
    Get-VM [[-Name] <String[]>] [-Server <VIMServer[]>] [-Datastore
    <StorageResource[]>] [-Location <VIMContainer[]>] [-Tag <Tag[]>]
    [-NoRecursion] [<CommonParameters>]

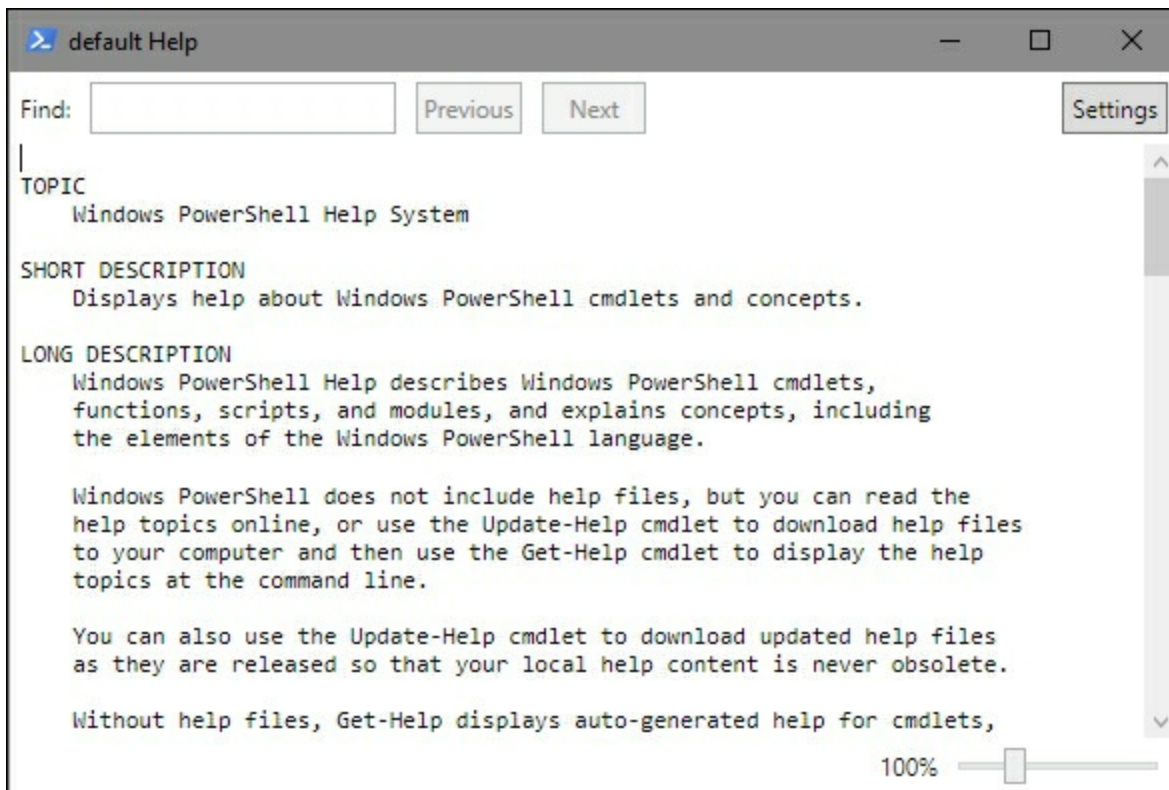
    Get-VM [[-Name] <String[]>] [-Server <VIMServer[]>]
    [-VirtualSwitch <VirtualSwitchBase[]>] [-Tag <Tag[]>]
    [<CommonParameters>]

    Get-VM [-Server <VIMServer[]>] -Id <String[]>
    [<CommonParameters>]

    Get-VM -RelatedObject <VmRelatedObjectBase[]>
    [<CommonParameters>]

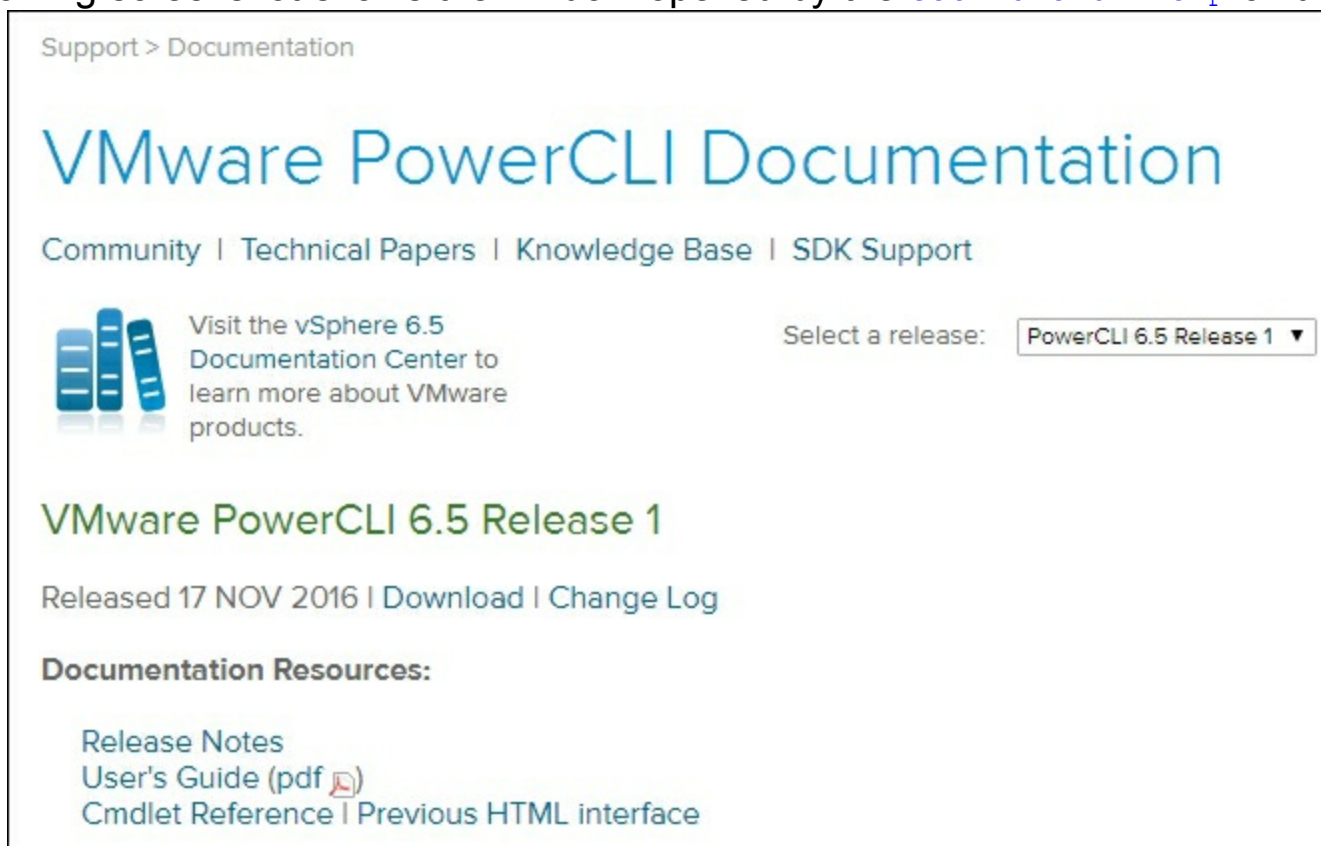
DESCRIPTION
    This cmdlet retrieves the virtual machines on a vCenter Server
    system. Returns a set of virtual machines that correspond to
    the filter criteria provided by the cmdlet parameters. For
    virtual machines with multiple NICs and multiple IP addresses,
    the IPAddress property of the VMGuest object contains all IP
    addresses of the virtual machine. The IP at position 0 is the
    primary IP address.
```

The `Get-Help` cmdlet has some parameters that you can use to get more information. The `-Examples` parameter shows examples of the cmdlet. The `-Detailed` parameter adds parameter descriptions and examples to the basic help display. The `-Full` parameter displays all the information available about the cmdlet. And the `-Online` parameter retrieves online help information available about the cmdlet and displays it in a web browser. Since PowerShell V3, there is a new `Get-Help` parameter `-ShowWindow`. This displays the output of `Get-Help` in a new window. The `Get-Help -ShowWindow` command opens the window in the following screenshot:



Using Get-PowerCLIHelp

The PowerCLI `Get-PowerCLIHelp` cmdlet opens the **VMware PowerCLI Documentation** website in a browser. You can find the PowerCLI documentation at the following URL: <https://www.vmware.com/support/developer/PowerCLI/>. You will have access to the **Release Notes**, **User's Guide**, and the **Cmdlet Reference | Previous HTML interface**. The following screenshot shows the window opened by the `Get-PowerCLIHelp` cmdlet:



Using Get-PowerCLICommunity

If you have a question about PowerCLI and you cannot find the answer in this book, use the

[Get-PowerCLICommunity](#) cmdlet to open the VMware PowerCLI section of the VMware VMTN Communities. You can log in to the VMware VMTN Communities using the same **My VMware** account that you used to download PowerCLI. First, search the community for an answer to your question. If you still cannot find the answer, go to the **Discussions** tab and ask your question by clicking on the **Start a Discussion** button, as shown later. You might receive an answer to your question in a few minutes.



Using Get-Member

In PowerCLI, you work with objects. Even a string is an object. An object contains properties and methods, which are called members in PowerShell. To see which members an object contains, you can use the [Get-Member](#) cmdlet. To see the members of a string, type the following command:

```
PowerCLI C:\> "Learning PowerCLI" | Get-Member
```

Pipe an instance of a PowerCLI object to [Get-Member](#) to retrieve the members of that PowerCLI object. For example, to see the members of a virtual machine object, you can use the following command:

```
PowerCLI C:\> Get-VM | Get-Member
```

The preceding command returns the following output:

```
TypeName:
VMware.VimAutomation.ViCore.Impl.V1.VM.UniversalVirtualMachineImpl

      Name                               MemberType Definition
-----
ConvertToVersion                         Method      T
VersionedObjectInterop.Conve...
Equals                                  Method      bool Equals(System.Object obj)
GetConnectionParameters                 Method
VMware.VimAutomation.ViCore.In...
GetHashCode                             Method      int GetHashCode()
GetType                                 Method      type GetType()
IsConvertibleTo                         Method      bool
VersionedObjectInterop.Is...
LockUpdates                             Method      void ExtensionData.LockUpdates()
ObtainExportLease                       Method
VMware.Vim.ManagedObjectRefere...
ToString                                Method      string ToString()
UnlockUpdates                           Method      void
ExtensionData.UnlockUpdat...
Client                                  Property
VMware.VimAutomation.ViCore.In...
CoresPerSocket                         Property    int CoresPerSocket {get;}
CustomFields                           Property
System.Collections.Generic.IDi...
DatastoreIdList                        Property    string[] DatastoreIdList {get;}
DrsAutomationLevel                     Property
System.Nullable[VMware.VimAuto...
```

ExtensionData	Property	System.Object ExtensionData
{g...		
Folder	Property	
VMware.VimAutomation.ViCore.Ty...		
FolderId	Property	string FolderId {get;}
Guest	Property	
VMware.VimAutomation.ViCore.Ty...		
GuestId	Property	string GuestId {get;}
HAIsolationResponse	Property	
System.Nullable[VMware.VimAuto...		
HARestartPriority	Property	
System.Nullable[VMware.VimAuto...		
Id	Property	string Id {get;}
MemoryGB	Property	decimal MemoryGB {get;}
MemoryMB	Property	decimal MemoryMB {get;}
Name	Property	string Name {get;}
Notes	Property	string Notes {get;}
NumCpu	Property	int NumCpu {get;}
PersistentId	Property	string PersistentId {get;}
PowerState	Property	
VMware.VimAutomation.ViCore.Ty...		
ProvisionedSpaceGB	Property	decimal ProvisionedSpaceGB
{get;}		
ResourcePool	Property	
VMware.VimAutomation.ViCore.Ty...		
ResourcePoolId	Property	string ResourcePoolId {get;}
Uid	Property	string Uid {get;}
UsedSpaceGB	Property	decimal UsedSpaceGB {get;}
VApp	Property	
VMware.VimAutomation.ViCore.Ty...		
Version	Property	
VMware.VimAutomation.ViCore.Ty...		
VMHost	Property	
VMware.VimAutomation.ViCore.Ty...		
VMHostId	Property	string VMHostId {get;}
VMResourceConfiguration	Property	
VMware.VimAutomation.ViCore.Ty...		
VMSwapfilePolicy	Property	
System.Nullable[VMware.VimAuto...		

The command returns the full type name of the `VirtualMachineImpl` object and all its methods and properties.

Tip

Remember that the properties are objects themselves. You can also use `Get-Member` to get the members of the properties. For example, the following command line will give you the members of the `VMGuestImpl` object:

```
PowerCLI C:\> $VM = Get-VM -Name vCenter
PowerCLI C:\> $VM.Guest | Get-Member
```

Using providers and PSDrives

Until now, you have only seen cmdlets. Cmdlets are PowerShell commands. PowerShell has another import concept named **providers**. Providers are accessed through named drives or **PSDrives**. In the following sections, Using providers and Using PSDrives, providers and PSDrives will be explained.

Using providers

A PowerShell provider is a piece of software that makes datastores look like filesystems. PowerShell providers are usually part of a snap-in or a module-like PowerCLI. The advantage of providers is that you can use the same cmdlets for all the providers. These cmdlets have the following nouns: `Item`, `ChildItem`, `Content`, and `ItemProperty`. You can use the `Get-Command` cmdlet to get a list of all the cmdlets with these nouns:

```
PowerCLI C:> Get-Command -Noun Item,ChildItem,Content,ItemProperty  
|  
Format-Table -AutoSize
```

The preceding command gives the following output:

CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Add-Content	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Clear-Content	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Clear-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Clear-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Copy-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Copy-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Get-ChildItem	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Get-Content	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Get-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Get-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Invoke-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Move-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Move-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	New-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	New-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Remove-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Remove-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Rename-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Rename-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Set-Content	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Set-Item	3.1.0.0	
Microsoft.PowerShell.Mana...			
Cmdlet	Set-ItemProperty	3.1.0.0	
Microsoft.PowerShell.Mana...			

To display a list of all the providers in your PowerCLI session, you can use the `Get-PSProvider` cmdlet:

```
PowerCLI C:\> Get-PSPProvider
Name                Capabilities                Drives
----                -
Alias                ShouldProcess                {Alias}
Environment          ShouldProcess                {Env}
FileSystem            Filter, ShouldProcess, Credentials {C, A, D, Z}
Function              ShouldProcess                {Function}
Registry              ShouldProcess, Transactions   {HKLM, HKCU}
Variable              ShouldProcess                {Variable}
VimDatastore          Filter, ShouldProcess        {vmstores, vmstore}
VimInventory          None                          {vis, vi}
```

The `VimDatastore` and `VimInventory` providers are part of PowerCLI. You will soon learn more about the `VimDatastore` and `VimInventory` providers.

Using PSDrives

Each provider has one or more drives. For example, the `FileSystem` provider has drives named `C`, `A`, `D`, and `Z`, which are hard disks on my PC. You can use the drives to access the providers. Microsoft calls these drives PSDrives to prevent confusing the drives with physical drives in your computer. For instance, to get a listing of all the files and folders in the root of `C:` on your PC, type the following command:

```
PowerCLI C:\> Get-ChildItem C:\
```

The `Get-ChildItem` cmdlet has aliases `dir`, `gci` and `ls` that give you the same result:

```
PowerCLI C:\> dir C:\
PowerCLI C:\> ls C:\
```

You can use the same `Get-ChildItem` cmdlet to get a list of all your cmdlet aliases by typing the following command:

```
PowerCLI C:\> Get-ChildItem alias:
```

The `Registry` PSDrive can be used to browse through the registry on your PC. The following command will list all the keys and values in the `HKEY_LOCAL_MACHINE\SOFTWARE` registry hive:

```
PowerCLI C:\> Get-ChildItem HKLM:\SOFTWARE
```

Using the PowerCLI Inventory Provider

The **Inventory Provider** gives you a filesystem-like view of the inventory items from a vCenter Server or an ESXi server. You can use this provider to view, move, rename, or delete objects by running PowerCLI commands.

When you connect to a server with the `Connect-VIServer` cmdlet, two PSDrives are created: `vi` and `vis`. The `vi` PSDrive contains the inventory of the last connected server. The `vis` PSDrive contains the inventory of all currently connected servers in your PowerCLI session.

You can set the location to the `vis` PSDrive using the `Set-Location` cmdlet:

```
PowerCLI C:\> Set-Location vis:
PowerCLI vis:\>
```

Use the `Get-ChildItem` cmdlet to display the items in the current location of the `vis`

PSDrive:

```
PowerCLI vis:\> Get-ChildItem
Name                               Type      Id
----                               -
192.168.0.132@443                 VIMServer
/VIMServer=vsphere.local\administrator@...
```

Use the `Get-ChildItem -Recurse` parameter to display all the items in the Inventory Provider:

```
PowerCLI vis:\> Get-ChildItem -Recurse
```

Using the PowerCLI Datastore Provider

The Datastore Provider gives you access to the content of your vSphere datastores. When you connect to a server with the `Connect-VIServer` cmdlet, two PSDrives are created: `vmstore` and `vmstores`. The `vmstore` PSDrive contains the datastores of the last connected server. The `vmstores` PSDrive contains the datastores of all the currently connected servers in your PowerCLI session. You can use these two default PSDrives or you can create custom PSDrives using the `New-PSDrive` cmdlet.

Set the location to the `vmstore` PSDrive with the following command:

```
PowerCLI C:\> Set-Location vmstore:
PowerCLI vmstore:\>
```

Display the content of the root directory of the `vmstore` PSDrive with the following command:

```
PowerCLI vmstore:\> Get-ChildItem
Name      Type      Id
----      -
New York  Datacenter Datacenter-datacenter-2
```

You can also create a custom PSDrive for a datastore using the `New-PSDrive` cmdlet. Start with getting a datastore object and save it in the `$Datastore` variable:

```
PowerCLI C:\> $Datastore = Get-Datastore -Name Datastore1
```

Create a new PowerShell PSDrive named `ds`, which maps to the `$Datastore` variable:

```
PowerCLI C:\> New-PSDrive -Location $Datastore -Name ds -PSProvider
VimDatastore -Root ""
```

Now, you can change your location into the PowerShell PSDrive using the `Set-Location` cmdlet:

```
PowerCLI C:\> Set-Location ds:
```

You can get a listing of the files and directories on the datastore using the `Get-ChildItem` cmdlet:

```
PowerCLI ds:\> Get-ChildItem
```

You will see an output similar to the following:

Name	Type	Id
----	----	--
DC1	DatastoreFolder	
.sdd.sf	DatastoreFolder	

Copying files between a datastore and your PC

You can use the vSphere Datastore Provider to copy files between a datastore and your PC using the `Copy-DatastoreItem` cmdlet.

Change the location to a subfolder using the `Set-Location` cmdlet with the help of the following command line:

```
PowerCLI ds:\> Set-Location "virtualmachine1"
```

Copy a file or directory to the destination using the `Copy-DatastoreItem` cmdlet, as follows:

```
PowerCLI ds:\virtualmachine1> Copy-DatastoreItem -Item
ds:\virtualmachine1\virtualmachine1.vmx -Destination
$env:USERPROFILE
```

Now, you can view the content of the `virtualmachine1.vmx` file with the following command:

```
PowerCLI C:\> Get-Content $env:USERPROFILE\virtualmachine1.vmx
```

`$env:USERPROFILE` is the path to your user profile, for example, `C:\users\username`.

Tip

Files cannot be copied directly between vSphere datastores in different vCenter Servers using `Copy-DatastoreItem`. Copy the files to the PowerCLI host's local filesystem temporarily and then copy them to the destination.

Using arrays and hash tables

In PowerCLI, you can create a list of objects. For example, `red`, `white`, and `blue` is a list of strings. In PowerShell, a list of terms is named an **array**. An array can have zero or more objects. You can create an empty array and assign it to a variable:

```
PowerCLI C:\> $Array = @()
```

You can fill the array during creation using the following command line:

```
PowerCLI C:\> $Array = @("red","white")
```

You can use the `+=` operator to add an element to an array:

```
PowerCLI C:\> $Array += "blue"
PowerCLI C:\> $Array
red
white
blue
```

If you want to retrieve a specific element of an array, you can use an index starting with `0` for the first element, `1` for the second element, and so on. If you want to retrieve an element from the tail of the array, you have to use `-1` for the last element, `-2` for the second to last, and so on. You have to use square brackets around the index number. In the next example, the first element of the array is retrieved using the following command line:

```
PowerCLI C:\> $Array[0]
Red
```

If you want to test if an object is an array, you can use the following command:

```
PowerCLI C:\> $Array -is [array]
True
```

There is a different kind of an array called a **hash table**. In a hash table, you map a set of keys to a set of values. You can create an empty hash table using the following command line:

```
PowerCLI C:\> $HashTable = @{}
```

You can fill the hash table during creation using the following command line:

```
PowerCLI C:\> $HashTable = @{LastName='Doe';FirstName='John'}
```

To add a key-value pair to a hash table, you can use the following command line:

```
PowerCLI C:\> $HashTable["Company"]='VMware'
```

To show the contents of the hash table, just display the variable:

```
PowerCLI C:\> $HashTable
Name                               Value
----                               -
Company                           VMware
FirstName                         John
LastName                          Doe
```

If you want to retrieve a specific key-value pair, you can use the following command:

```
PowerCLI C:\> $HashTable["FirstName"]  
John
```

To retrieve all of the hash table's keys, you can use the `Keys` property:

```
PowerCLI C:\> $HashTable.Keys  
Company  
FirstName  
LastName
```

To retrieve all the values in the hash table, you can use the `Values` property:

```
PowerCLI C:\> $HashTable.Values  
VMware  
John  
Doe
```

If you want to test whether an object is a hash table, you can use the following command:

```
PowerCLI C:\> $HashTable -is [hashtable]  
True
```

In the next section, hash tables will be used to create calculated properties.

Creating calculated properties

You can use the `Select-Object` cmdlet to select certain properties of the objects that you want to return. For example, you can use the following code to return the name and the used space, in GB, of your virtual machines:

```
PowerCLI C:\> Get-VM | Select-Object -Property Name,UsedSpaceGB
```

But what if you want to return the used space in MB? The PowerCLI `VirtualMachineImpl` object has no `UsedSpaceMB` property. This is where you can use a **calculated property**. A calculated property is a PowerShell hash table with two elements: `Name` and `Expression`. The `Name` element contains the name that you want to give the calculated property. The `Expression` element contains a scriptblock with PowerCLI code to calculate the value of the property. To return the name and the used space in MB for all your virtual machines, run the following command:

```
PowerCLI C:\> Get-VM |  
>> Select-Object -Property Name,  
>> @{Name="UsedSpaceMB";Expression={1KB*$_UsedSpaceGB}}  
>>
```

The hash table contains two key-value pairs:

- In the first element, the key is `Name` and the value is `UsedSpaceMB`
- In the other element, the key is `Expression`, and the value is `{1KB*$_UsedSpaceGB}`

The special variable `$_` is used to represent the current object in the pipeline. `1KB` is a PowerShell constant that has the value `1024`. In calculated properties, you can abbreviate the `Name` and `Expression` names to `N` and `E`.

Another example of a calculated property shows you how to return the aliases of all the cmdlets that are the same for all the providers:

```
PowerCLI C:\> Get-Command -Noun Item,ChildItem,Content,ItemProperty
|
>> Select-Object -Property Name,
>> @{Name="Aliases";Expression={Get-Alias -Definition $_.Name}}
>>
```

Name	Aliases
----	-----
Add-Content	ac
Clear-Content	clc
Clear-Item	cli
Clear-ItemProperty	clp
Copy-Item	{copy, cp, cpi}
Copy-ItemProperty	cnp
Get-ChildItem	{dir, gci, ls}
Get-Content	{cat, gc, type}
Get-Item	gi
Get-ItemProperty	gp
Invoke-Item	ii
Move-Item	{mi, move, mv}
Move-ItemProperty	mp
New-Item	ni
New-ItemProperty	
Remove-Item	{del, erase, rd, ri...}
Remove-ItemProperty	rp
Rename-Item	{ren, rni}
Rename-ItemProperty	rnnp
Set-Content	sc
Set-Item	si
Set-ItemProperty	sp

The first command is the `Get-Command` statement that you have seen before; this returns the cmdlets that are the same for all the providers. In the calculated property, the `Get-Alias` cmdlet is used to get the aliases of these commands.

Using raw API objects with ExtensionData or Get-View

PowerCLI makes it easy to use the VMware vSphere **application programming interface (API)**. There are two ways to do this. The first one is by using the `ExtensionData` property that most of the PowerCLI objects have. The `ExtensionData` property is a direct link to the vSphere API object related to the PowerCLI object. The second way is by using the `Get-View` cmdlet to retrieve the vSphere API object related to a PowerCLI object. Both these ways will be discussed in the following sections.

Using the ExtensionData property

Most PowerCLI objects, such as `VirtualMachineImpl` and `VMHostImpl`, have a property named `ExtensionData`. This property is a reference to a view of a VMware vSphere object as described in the VMware vSphere API Reference documentation. For example, the `ExtensionData` property of the PowerCLI's `VirtualMachineImpl` object links to a vSphere `VirtualMachine` object view. `ExtensionData` is a very powerful property because it allows you to use all the properties and methods of the VMware vSphere API. For example, to check whether the VMware Tools are running in your virtual machines, you can run the following command:

```
PowerCLI C:\> Get-VM |
>> Select-Object -Property Name,
>> @{Name = "ToolsRunningStatus";
>>     Expression = {$_ .ExtensionData.Guest.ToolsRunningStatus}
>> }
>>
```

If VMware Tools are not installed in a virtual machine, the `ExtensionData.Guest.ToolsStatus` property will have the value `toolsNotInstalled`. You can check the tool's status with the following command:

```
PowerCLI C:\> Get-VM |
>> Select-Object -Property Name,
>> @{Name = "ToolsStatus"
>>     Expression = {$_ .ExtensionData.Guest.ToolsStatus}
>> }
>>
```

Name	ToolsStatus
-----	-----
VM1	toolsNotInstalled
DNS1	toolsOk
DC1	toolsNotRunning
WindowsServer2012	toolsOld

Using the Get-View cmdlet

Another way to get the vSphere API objects is by using the `Get-View` cmdlet. This cmdlet returns a vSphere object view, which is the same object you can retrieve via the `ExtensionData` property. For example, the following two PowerCLI commands will give you

the same result:

```
PowerCLI C:\> (Get-VM -Name vCenter).ExtensionData
PowerCLI C:\> Get-View -VIOBJECT (Get-VM -Name vCenter)
```

The `Get-View` cmdlet has the following syntax:

```
Get-View [-VIOBJECT] <VIOBJECT[]> [-Property <String[]>]
    [<CommonParameters>]
Get-View [-Server <VIServer[]>] [-Id] <ManagedObjectReference[]>
    [-Property <String[]>] [<CommonParameters>]
Get-View [-Server <VIServer[]>] [-SearchRoot
    <ManagedObjectReference>]
    -ViewType <Type> [-Filter <Hashtable>] [-Property <String[]>]
    [<CommonParameters>]
Get-View [-Property <String[]>] -RelatedObject
    <ViewBaseMirroredObject[]> [<CommonParameters>]
```

The names of the parameter sets are `GetViewByVIOBJECT`, `GetView`, `GetEntity`, and `GetViewByRelatedObject`. The third parameter set, `GetEntity`, is very powerful and will allow you to create PowerCLI commands or scripts that are optimized for speed. For example, the following command will give you the vSphere object views of all virtual machines and templates:

```
PowerCLI C:\> Get-View -ViewType VirtualMachine
```

Possible argument values for the `-ViewType` parameter are `ClusterComputeResource`, `ComputeResource`, `Datacenter`, `Datastore`, `DistributedVirtualPortgroup`, `DistributedVirtualSwitch`, `Folder`, `HostSystem`, `Network`, `OpaqueNetwork`, `ResourcePool`, `StoragePod`, `VirtualApp`, `VirtualMachine`, and `VmwareDistributedVirtualSwitch`. If you require only the virtual machines and not the templates, you need to specify a filter:

```
PowerCLI C:\> Get-View -ViewType VirtualMachine -Filter
    @{" Config.Template" = "false"}
```

The filter is in the form of a hash table in which you specify that the value of the `Config.Template` property needs to be false to get only the virtual machines. To make your command run faster, you need to specify the properties that you want to return. Otherwise, all the properties are returned, and it will make your command run slower.

Let's retrieve only the name and the overall status of your virtual machines:

```
PowerCLI C:\> Get-View -ViewType VirtualMachine -Filter
    @{"Config.Template" = "false"} -Property Name,OverallStatus |
>> Select-Object -Property Name,OverAllStatus
>>
```

This command runs in my test environment about 23 times faster than the equivalent:

```
PowerCLI C:\> Get-VM | Select-Object -Property Name,
    @{"Name="OverallStatus";Expression={$_.ExtensionData.OverallStatus}}
```

The conclusion is if you need your script to run faster, try to find a solution using the `Get-View` cmdlet.

Tip

You should always make a trade-off between the time it takes you to write a script and the time it takes you to run the script. If you spend 10 minutes to create a script that takes 1 hour to run, you will have your work done in 70 minutes. If you spend 2 hours to create a faster script that runs in 10 minutes, you will have your work done in 130 minutes. I would prefer the first solution. Of course, if you intend to run the script more than once, the time you spend to improve the speed of your script is spent better.

Using managed object references

If you look at a vSphere object view using the `Get-Member` cmdlet, you will see that a lot of properties are from the type `VMware.Vim.ManagedObjectReference`:

```
PowerCLI C:\> Get-VM -Name vCenter | Get-View | Get-Member |
>> Where-Object {$_.Name -eq 'Parent'}
>>
```

```
TypeName: VMware.Vim.VirtualMachine
```

Name	MemberType	Definition
Parent	Property	VMware.Vim.ManagedObjectReference Parent {get;}

A **Managed Object Reference (MoRef)** is a unique value that is generated by the vCenter Server and is guaranteed to be unique for a given entity in a single vCenter instance.

Tip

The vSphere object views returned by the `ExtensionData` property or the `Get-View` cmdlet are not the actual vSphere objects. The objects returned are copies or views of the actual objects that represent the actual objects at the time the view was made.

Using the Get-VIObjectByVIView cmdlet

The `Get-View` cmdlet gives you a way to go from a PowerCLI object to a vSphere object view. If you want to go back from a vSphere object view to a PowerCLI object, you can use the `Get-VIObjectByVIView` cmdlet. Take a look at the following example:

```
PowerCLI C:\> $VMView = Get-VM -Name vCenter | Get-View
PowerCLI C:\> $VM = $VMView | Get-VIObjectByVIView
```

In the preceding example, the first line will give you a vSphere object view from a PowerCLI `VirtualMachineImpl` object. The second line will convert the vSphere object view back to a PowerCLI `VirtualMachineImpl` object.

The `Get-VIObjectByVIView` cmdlet has the following syntax:

```
Get-VIObjectByVIView [-VIView] <ViewBase[]> [<CommonParameters>]
Get-VIObjectByVIView [-Server <VIServer[]>] [-MORef]
    <ManagedObjectReference[]> [<CommonParameters>]
```

You can see that the `Get-VIObjectByVIView` cmdlet has two parameter sets. The first parameter set contains the `-VIView` parameter. The second parameter set contains the `-Server` and `-MORef` parameters.

Note

Remember that parameters from different parameter sets cannot be mixed in one command.

If you are connected to multiple vCenter Servers, the `Get-VIObjectByVIView` cmdlet might return objects from multiple vCenter Servers because MoRefs are only unique on a single vCenter Server instance. You can use the `-Server` parameter of the `Get-VIObjectByVIView` cmdlet to solve this problem by specifying the vCenter Server for which you want to return objects. Because the `-Server` parameter is in another parameter set and not in the `-VIView` parameter, you cannot use the `-VIView` parameter that is used in the pipeline. You have to use the `ForEach-Object` cmdlet and the `-MORef` parameter of the `Get-VIObjectByVIView` cmdlet:

```
PowerCLI C:\> $VMView |  
>> ForEach-Object {  
>>     Get-VIObjectByVIView -Server vCenter1 -MORef $_.MORef  
>> }  
>>
```

Note

In the name of the `Get-VIObjectByVIView` cmdlet, you can see a piece of the history of PowerCLI. VMware vSphere was named VMware Infrastructure before VMware vSphere 4. The earlier VMware PowerCLI versions were named VI Toolkit. In the name of this cmdlet, you see that a PowerCLI object is still named a `VIObject` and a vSphere object view is named a `VIView`.

Extending PowerCLI objects with the New-VIProperty cmdlet

Sometimes, you can have the feeling that a PowerCLI object is missing a property. Although the VMware PowerCLI team tried to include the most useful properties in the objects, you can have the need for an extra property. Luckily, PowerCLI has a way to extend a PowerCLI object using the `New-VIProperty` cmdlet. This cmdlet has the following syntax:

```
New-VIProperty [-Name] <String> [-ObjectType] <String[]> [-Value]
    <ScriptBlock> [-Force] [-BasedOnExtensionProperty <String[]>]
[-WhatIf]
    [-Confirm] [<CommonParameters>]
New-VIProperty [-Name] <String> [-ObjectType] <String[]> [-Force]
    [-ValueFromExtensionProperty] <String> [-WhatIf] [-Confirm]
    [<CommonParameters>]
```

Let's start with an example. You will add the VMware Tools' running statuses used in a previous example to the `VirtualMachineImpl` object using the `New-VIProperty` cmdlet:

```
PowerCLI C:\> New-VIProperty -ObjectType VirtualMachine -Name
    ToolsRunningStatus -ValueFromExtensionProperty
    'Guest.ToolsRunningStatus'
```

Name	RetrievingType	DeclaringType	Value
ToolsRunning...	VirtualMachine	VirtualMachine	
Guest.ToolsRunningStatus			

Now you can get the tools' running statuses of all of your virtual machines with the following command:

```
PowerCLI C:\> Get-VM | Select-Object -Property Name,
    ToolsRunningStatus
```

Isn't this much easier?

In the next example, you will add the `vCenterServer` property to the `VirtualMachineImpl` object. The name of the vCenter Server is part of the `VirtualMachineImpl Uid` property. The `Uid` property is a string that looks like

`/VIServer=domain\account@vCenter:443/VirtualMachine=VirtualMachine-vm-239/`.

You can use the `Split()` method to split the string. For example, the following command splits the string `192.168.0.1` at the dots into an array with four elements:

```
PowerCLI C:\> "192.168.0.1".Split('.')
192
168
0
1
```

The first element is `192`, the second element is `168`, the third element is `0`, and the fourth and last element is `1`. If you assign the array to a variable, then you can use an index to specify a certain element of the array:

```
PowerCLI C:\> $Array = "192.168.0.1".Split('.')
```

The index is `0` for the first element, `1` for the second element, and so on. If you want to specify the last element of the array, you can use the index `-1`. Take a look at the following example:

```
PowerCLI C:\> $Array[0]
192
```

In the `Uid` property, the name of the vCenter Server is between the `@` sign and the colon. So, you can use those two characters to split the string. First, you split the string at the colon and take the part before the colon. That is the first element of the resulting array:

```
PowerCLI C:\> $Uid =
'/VIServer=domain\account@vCenter:443/VirtualMachine=
VirtualMachine-vm-239/'
PowerCLI C:\> $Uid.Split(':')[0]
/VIServer=domain\account@vCenter
```

Split the resulting part at the `@` sign and take the second element of the resulting array to get the name of the vCenter Server:

```
PowerCLI C:\> $String = '/VIServer=domain\account@vCenter'
PowerCLI C:\> $String.Split('@')[1]
vCenter
```

You can do this splitting with one line of code:

```
PowerCLI C:\> $Uid =
'/VIServer=domain\account@vCenter:443/VirtualMachine=
VirtualMachine-vm-239/'
PowerCLI C:\> $Uid.Split(':')[0].Split('@')[1]
vCenter
```

Use the `-Value` parameter of the `New-VIProperty` cmdlet to specify a scriptblock. In this scriptblock, `$Args[0]` is the object with which you want to retrieve the name of the vCenter Server:

```
PowerCLI C:\> New-VIProperty -Name vCenterServer -ObjectType
VirtualMachine -Value {$Args[0].Uid.Split(":")[0].Split("@")
[1]} -Force
```

The `New-VIProperty -Force` parameter indicates that you want to create the new property even if another property with the same name already exists for the specified object type. Now you can get a list of all of your virtual machines and their vCenter Servers with the following command:

```
PowerCLI C:\> Get-VM | Select-Object -Property Name,vCenterServer
```

Working with vSphere folders

In a VMware vSphere environment, you can use folders to organize your infrastructure. In the vSphere web client, you can create folders in the [Hosts and Clusters](#), [VMs and Templates](#), [Storage](#), and [Networking](#) inventories. The following screenshot shows an example of folders in the [VMs and Templates](#) inventory:



You can browse through these folders using the PowerCLI Inventory Provider. PowerCLI also has a set of cmdlets to work with these folders: [Get-Folder](#), [Move-Folder](#), [New-Folder](#), [Remove-Folder](#), and [Set-Folder](#).

You can use the [Get-Folder](#) cmdlet to get a list of all of your folders:

```
PowerCLI C:\> Get-Folder
```

Otherwise, you can select specific folders by their name using the following command line:

```
PowerCLI C:\> Get-Folder -Name "Accounting"
```

All folders are organized in a tree structure under the root folder. You can retrieve the root folder with the following command:

```
PowerCLI C:\> Get-Folder -NoRecursion
```

Name	Type
-----	-----
Datacenters	Datacenter

The root folder is always named [Datacenters](#). In this folder, you can only create subfolders or data centers.

Folders in vSphere are of a certain type. Valid folder types are [VM](#), [HostAndCluster](#), [Datastore](#), [Network](#), and [Datacenter](#). You can use this to specify the type of folders you want to retrieve. For example, to retrieve only folders of type [VM](#), use the following command:

```
PowerCLI C:\> Get-Folder -Type VM
```

A problem with folders is that you don't get the full path from the root if you retrieve a folder. Using the [New-VIProperty](#) cmdlet, you can easily add a [Path](#) property to a PowerCLI [Folder](#) object:

```

PowerCLI C:\> New-VIProperty -Name Path -ObjectType Folder -Value {
    # $FolderView contains the view of the current folder object
    $FolderView = $Args[0].Extensiondata

    # $Server is the name of the vCenter Server
    $Server = $Args[0].Uid.Split(":")[0].Split("@")[1]

    # We build the path from the right to the left
    # Start with the folder name
    $Path = $FolderView.Name

    # While we are not in the root folder
    while ($FolderView.Parent){
        # Get the parent folder
        $FolderView = Get-View -Id $FolderView.Parent -Server $Server

        # Extend the path with the name of the parent folder
        $Path = $FolderView.Name + " " + $Path
    }

    # Return the path
    $Path
} -Force # Create the property even if a property with this name
exists

```

In this example, you see that the # character in PowerShell is used to comment. Using the new `Path` property, you can now get the path for all the folders with the following command:

```
PowerCLI C:\> Get-Folder | Select-Object -Property Name,Path
```

You can use the `Path` property to find a folder by its complete path. Take a look at the following example:

```

PowerCLI C:\> Get-Folder |
>> Where-Object {$_.Path -eq 'Datacenters\Dallas\vm\Templates'}
>>

```

Summary

In this chapter, you looked at the `Get-Help`, `Get-Command`, and `Get-Member` cmdlets. You learned how to use providers and PSDrives. You also saw how to create a calculated property. Using the raw API objects with the `ExtensionData` property or the `Get-View` cmdlet was discussed, and you looked at extending PowerCLI objects with the `New-VIProperty` cmdlet. At the end, you learned to work with folders, and you saw how you can use the `New-VIProperty` cmdlet to extend the `Folder` object of PowerCLI with a `Path` property.

In the next chapter, you will learn more about working with objects in PowerCLI.

Chapter 3. Working with Objects in PowerShell

PowerShell is an object-oriented shell. Don't let this scare you because if you know how to work with PowerShell objects, it will make your life much easier. Objects in PowerShell have properties and methods, just like objects in real life. For example, let's take a computer and try to see it as an object. It has properties such as the manufacturer, the number of CPUs, the amount of memory, and the type of computer (for example, server, workstation, desktop, or laptop). The computer also has methods, for example, you can switch the computer on and off. Properties and methods together are called **members** in PowerShell. In [Chapter 2](#), Learning Basic PowerShell Concepts, you already saw the `Get-Member` cmdlet that lists the properties and methods of a PowerShell object. In this chapter, you will learn all of the ins and outs of PowerShell objects. We will focus on the following topics:

- Using objects, properties, and methods
- Expanding variables and subexpressions in strings
- Using here-strings
- Using the pipeline
- Using the PowerShell object cmdlets
- Creating your own objects
- Using COM objects

Using objects, properties, and methods

In PowerShell, even a string is an object. You can list the members of a string object using the `Get-Member` cmdlet that you have seen before. Let's go back to our example from [Chapter 2](#), Learning Basic PowerShell Concepts. First, we create a string `Learning PowerShell` and put it in a variable named `$String`. Then, we take the `$String` variable and execute the `Get-Member` cmdlet using the `$String` variable as the input:

```
PowerCLI C:\> $String = "Learning PowerShell"
PowerCLI C:\> Get-Member -InputObject $String
```

You can also use the pipeline and do it in a one-liner:

```
PowerCLI C:\> "Learning PowerShell" | Get-Member
```

The output will be as follows:

Name	MemberType	Definition
Clone	Method	System.Object Clone(),
Syst...		
CompareTo	Method	int
CompareTo(System.Object...)		
Contains	Method	bool Contains(string value)
CopyTo	Method	void CopyTo(int
sourceIndex...		
EndsWith	Method	bool EndsWith(string

value)...		
Equals	Method	bool Equals(System.Object
O...		
GetEnumerator	Method	System.CharEnumerator
GetEn...		
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode
GetTypeCode...		
IndexOf	Method	int IndexOf(char value),
in...		
IndexOfAny	Method	int IndexOfAny(char[]
anyOf...		
Insert	Method	string Insert(int
startInde...		
IsNormalized	Method	bool IsNormalized(), bool
I...		
LastIndexOf	Method	int LastIndexOf(char
value)...		
LastIndexOfAny	Method	int LastIndexOfAny(char[]
a...		
Normalize	Method	string Normalize(), string
...		
PadLeft	Method	string PadLeft(int
totalWid...		
PadRight	Method	string PadRight(int
totalWi...		
Remove	Method	string Remove(int
startInde...		
Replace	Method	string Replace(char
oldChar...		
Split	Method	string[] Split(Params
char[...		
StartsWith	Method	bool StartsWith(string
valu...		
Substring	Method	string Substring(int
startI...		
ToBoolean	Method	bool
IConvertible.ToBoolean...		
ToByte	Method	byte
IConvertible.ToByte(Sy...		
ToChar	Method	char
IConvertible.ToChar(Sy...		
ToCharArray	Method	char[] ToCharArray(),
char[...		
ToDateTime	Method	datetime
IConvertible.ToDat...		
ToDecimal	Method	decimal
IConvertible.ToDeci...		
ToDouble	Method	double
IConvertible.ToDoubl...		
ToInt16	Method	int16
IConvertible.ToInt16(...		
ToInt32	Method	int
IConvertible.ToInt32(Sy...		
ToInt64	Method	long

<code>IConvertible.ToInt64(S...</code>		
<code>ToLower</code>	Method	<code>string ToLower(), string</code>
<code>To...</code>		
<code>ToLowerInvariant</code>	Method	<code>string ToLowerInvariant()</code>
<code>ToSByte</code>	Method	<code>sbyte</code>
<code>IConvertible.ToSByte(...</code>		
<code>ToSingle</code>	Method	<code>float</code>
<code>IConvertible.ToSingle...</code>		
<code>ToString</code>	Method	<code>string ToString(), string</code>
<code>T...</code>		
<code>GetType</code>	Method	<code>System.Object</code>
<code>IConvertible....</code>		
<code>ToUInt16</code>	Method	<code>uint16</code>
<code>IConvertible.ToUInt1...</code>		
<code>ToUInt32</code>	Method	<code>uint32</code>
<code>IConvertible.ToUInt3...</code>		
<code>ToUInt64</code>	Method	<code>uint64</code>
<code>IConvertible.ToUInt6...</code>		
<code>ToUpper</code>	Method	<code>string ToUpper(), string</code>
<code>To...</code>		
<code>ToUpperInvariant</code>	Method	<code>string ToUpperInvariant()</code>
<code>Trim</code>	Method	<code>string Trim(params char[]</code>
<code>t...</code>		
<code>TrimEnd</code>	Method	<code>string TrimEnd(params</code>
<code>char[...</code>		
<code>TrimStart</code>	Method	<code>string TrimStart(params</code>
<code>cha...</code>		
<code>Chars</code>	ParameterizedProperty	<code>char Chars(int index) {get;}</code>
<code>Length</code>	Property	<code>int Length {get;}</code>

You may see that a string has a lot of methods, one property, and a special type of property named `ParameterizedProperty`. Let's first use the `Length` property. To use a property, type the object name or the name of the variable containing the object, then type a dot, and finally type the property name. So, for the string, you could use any of the following command lines:

```
PowerCLI C:\> "Learning PowerCLI".Length
17
```

Or:

```
PowerCLI C:\> $String.Length
17
```

You may see that the `Length` property contains the number of characters of the string `Learning PowerCLI`; 17 in this case.

Property names in PowerShell are not case-sensitive. So, you could type the following command as well:

```
PowerCLI C:\> $String.length
17
```

`ParameterizedProperty` is a property that accepts a parameter value. The `ParameterizedProperty char Chars` property can be used to return the character at a specific position in the string. You have to specify the position, also named the **index**, as a parameter to `Chars`. Indexes in PowerShell start with 0. So, to get the first character of the

string, type the following command:

```
PowerCLI C:\> $String.Chars(0)
L
```

To get the second character of the string, type the following command:

```
PowerCLI C:\> $String.Chars(1)
e
```

You cannot use `-1` to get the last character of the string, as you can do with indexing in a PowerShell array. You have to calculate the last index yourself, and it is calculated by subtracting `1` from the length of the string. So, to get the last character of the string, you can type the following command:

```
PowerCLI C:\> $String.Chars($String.Length - 1)
I
```

PowerShell has more types of properties, such as `AliasProperty`, `CodeProperty`, `NoteProperty`, and `ScriptProperty`:

- `AliasProperty` is an alias name for an existing property
- `CodeProperty` is a property that maps to a static method on a .NET class
- `NoteProperty` is a property that contains data
- `ScriptProperty` is a property whose value is returned from executing a PowerShell scriptblock

Using methods

Using `methods` is as easy as using properties. You can type the name of a variable containing the object, then you type a dot, and after the dot, you type the name of the method. For methods, you always have to use parentheses after the method name. For example, to modify a string to all uppercase letters type in the following command:

```
PowerCLI C:\> $String.ToUpper()
LEARNING POWERCLI
```

Some methods require parameters. For example, to find the index of the `P` character in the string, you can use the following command:

```
PowerCLI C:\> $String.IndexOf('P')
9
```

The character `P` is the tenth character in the `Learning PowerCLI` string. But because indexes in PowerShell start with `0` and not `1`, the index of the `P` character in the string is `9` and not `10`.

A very useful method is `Replace` that you can use to replace a character or a substring with another character, string, or nothing. For example, let's replace all `e` characters in the string with a `u` character:

```
PowerCLI C:\> $String.Replace('e','u')
Luarning PowurCLI
```

The characters in the method are case-sensitive. If you use an uppercase `E`, it won't find the letter and will replace nothing. See the following command:

```
PowerCLI C:\> $String.Replace('E','U')
Learning PowerCLI
```

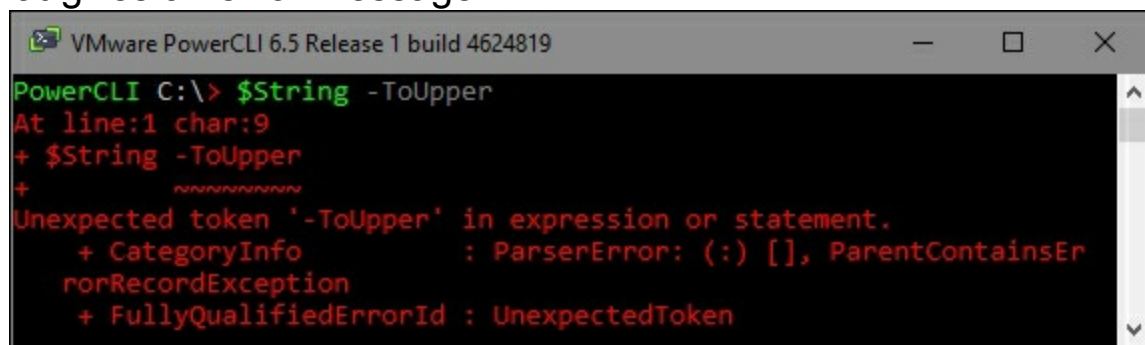
You can also replace a substring with another string. Let's replace the word `PowerCLI` with `VMware PowerCLI`:

```
PowerCLI C:\> $String.Replace('PowerCLI','VMware PowerCLI')
Learning VMware PowerCLI
```

There is also a `-Replace` operator in PowerShell. You can use the `-Replace` operator to do a regular expression-based text substitution on a string or a collection of strings:

```
PowerCLI C:\> $string -Replace 'e','u'
Luarning PowurCLI
```

Although both have the same name, the string `Replace` method and the `-Replace` operator are two different things. There is no `-ToUpper` operator, as you can see in the following screenshot that gives an error message:

A screenshot of a PowerShell console window titled "VMware PowerCLI 6.5 Release 1 build 4624819". The command prompt shows the command `$String -ToUpper`. The output indicates an error: "At line:1 char:9", "+ \$String -ToUpper", "+", "Unexpected token '-ToUpper' in expression or statement.", "+ CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException", and "+ FullyQualifiedErrorId : UnexpectedToken".

```
VMware PowerCLI 6.5 Release 1 build 4624819
PowerCLI C:\> $String -ToUpper
At line:1 char:9
+ $String -ToUpper
+
Unexpected token '-ToUpper' in expression or statement.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : UnexpectedToken
```

You can use more than one method in the same command. Say, you want to replace the word `Learning` with `Gaining`, and that you want to remove the characters `C`, `L`, and `I` from the end of the string using the `TrimEnd` method. Then, you can use the following command:

```
PowerCLI C:\> $String.Replace('Learning','Gaining').TrimEnd('CLI')
Gaining Power
```

Expanding variables and subexpressions in strings

In PowerShell, you can define a string with single or double quotes. There is a difference between these two methods. In a single-quoted string, variables and subexpressions are not expanded, whereas, in a double-quoted string, they are expanded.

Let's look at an example of variable expansion in a double-quoted string:

```
PowerCLI C:\> $Number = 3
PowerCLI C:\> "The number is: $Number"
The number is: 3
```

In the preceding example, the string is defined with double quotes, and the `$Number` variable is expanded. Let's see what happens if you use single quotes:

```
PowerCLI C:\> $Number = 3
PowerCLI C:\> 'The number is: $Number'
The number is: $Number
```

Using a single-quoted string, PowerShell doesn't expand the `$Number` variable. Let's try to put the number of virtual CPUs of a virtual machine in a double-quoted string:

```
PowerCLI C:\> $vm = Get-VM -Name dc1
PowerCLI C:\> "The number of vCPU's of the vm is: $vm.NumCpu"
The number of vCPU's of the vm is: dc1.NumCpu
```

The output is not what you intended. What happened? The `$` sign in front of a variable name tells PowerShell to evaluate the variable. In the string that is used in the preceding example, `$vm` evaluates the variable `vm`. But it does not evaluate `$vm.NumCpu`. To evaluate `$vm.NumCpu`, you have to use another `$` sign before and parentheses around the code that you want to evaluate: `$($vm.NumCpu)`. This is called a **subexpression** notation.

In the corrected example, you will get the number of virtual CPUs:

```
PowerCLI C:\> $vm = Get-VM -Name dc1
PowerCLI C:\> "The number of vCPU's of the vm is: $($vm.NumCpu)"
The number of vCPU's of the vm is: 2
```

You can use subexpression evaluation to evaluate any PowerShell code. In the following example, you will use PowerShell to calculate the sum of 3 and 4:

```
PowerCLI C:\> "3 + 4 = $(3+4)"
3 + 4 = 7
```

When will a string be expanded?

A string will be expanded when it is assigned to a variable. It will not be re-evaluated when the variable is used later. The following example shows this behavior:

```
PowerCLI C:\> $Number = 3
PowerCLI C:\> $String = "The number is: $Number"
PowerCLI C:\> $String
The number is: 3
PowerCLI C:\> $Number = 4
PowerCLI C:\> $String
The number is: 3
```

As you can see, `$String` is assigned before `$Number` gets the value 4. The `$String` variable says `The number is: 3`.

Expanding a string when it is used

How can you delay the expansion of the string until you use it? PowerShell has a predefined variable named `$ExecutionContext`. You can use the `InvokeCommand.ExpandString()` method of this variable to expand the string:

```
PowerCLI C:\> $Number = 3
PowerCLI C:\> $String = 'The number is: $Number'
PowerCLI C:\> $ExecutionContext.InvokeCommand.ExpandString($String)
The number is: 3
PowerCLI C:\> $Number = 4
PowerCLI C:\> $ExecutionContext.InvokeCommand.ExpandString($String)
The number is: 4
```

The preceding example defines `$String` as a single-quoted string, so `$Number` is not expanded at the assignment of `$String`. The

`$ExecutionContext.InvokeCommand.ExpandString($String)` command expands the string every time the command is executed.

Using here-strings

Until now, you have only seen single-line strings in this book. PowerShell has a so-called **here-string** that spans multiple lines. You use `@"` or `@'` to start the here-string and `"@` or `'@` to finish the here-string. The `@"` or `@'` must be at the end of a line and the `"@` or `'@` must be at the beginning of the line that terminates the here-string. As in single-line strings, variables and subexpressions are expanded in double-quoted here-strings and are not expanded in single-quoted here-strings.

The following command creates a here-string that spans two lines and puts the here-string in the `$s` variable:

```
PowerCLI C:\> $s = @"
>> Learning PowerCLI
>> is a lot of fun!
>> "@
>> $s
>>
Learning PowerCLI
is a lot of fun!
```

Using the pipeline

In PowerShell, you can use the output of one command as input for another command by using the vertical bar (|) character. This is called using the pipeline. The vertical bar character, in PowerShell, is called the **pipe** character. In PowerShell, complete objects pass through the pipeline. This is different from `cmd.exe` or a Linux shell where only strings pass through the pipeline. The advantage of passing complete objects through the pipeline is that you don't have to perform string manipulations to retrieve property values.

Using the ByValue parameter binding

You have already seen some examples of using the pipeline in preceding sections of this book. For example:

```
PowerCLI C:\> Get-VM | Get-Member
```

In this example, the output of the `Get-VM` cmdlet is used as the input for the `Get-Member` cmdlet. This is much simpler than the following command, which gives you the same result:

```
PowerCLI C:\> Get-Member -InputObject (Get-VM)
```

You can see that the `Get-Member` cmdlet accepts inputs from the pipeline if you look at the help for the `Get-Member -Parameter InputObject` using the following command:

```
PowerCLI C:\> Get-Help Get-Member -Parameter InputObject
```

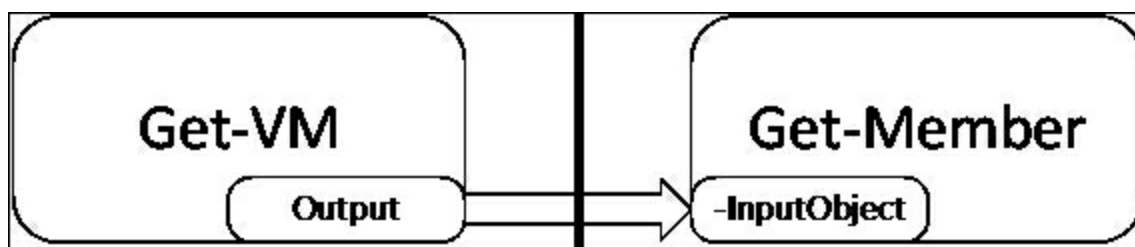
The output will be the following:

```
-InputObject <PSObject>
Specifies the object whose members are retrieved.
Using the InputObject parameter is not the same as piping an object
to Get-Member. The differences are as follows:
-- When you pipe a collection of objects to Get-Member, Get-Member
gets the members of the individual objects in the collection,
such as
the properties of each string in an array of strings.
-- When you use InputObject to submit a collection of objects,
Get-Member gets the members of the collection, such as the
properties of the array in an array of strings.
```

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

You can see that in the description it says `Accept pipeline input? true (ByValue)`. This means that the `Get-Member -InputObject` parameter accepts inputs from the pipeline. The `ByValue` parameter binding means that PowerShell binds the entire input object to the parameter.

In the following figure, you will see that the pipeline binds the output of the `Get-Member` cmdlet to the `-InputObject` parameter of the `Get-Member` cmdlet:



Using the ByPropertyName parameter binding

If PowerShell can't find a parameter that accepts pipeline inputs using `ByValue`, it tries to find parameters that accept pipeline inputs using `ByPropertyName`. When a parameter accepts pipeline inputs using `ByPropertyName`, it means that the value of a property of the input object is bound to a cmdlet parameter with the same name as the property.

An example of a PowerShell cmdlet that accepts inputs from the pipeline using the `ByPropertyName` parameter binding is the `Get-Date` cmdlet that returns a `System.DateTime` object. The `-Date` parameter of this cmdlet accepts pipeline inputs using both `ByValue` and `ByPropertyName`. The PowerCLI `Get-VIEvent` cmdlet retrieves information about the events on a vCenter Server system. Take a look at the following example:

```
PowerCLI C:\> Get-VIEvent | Select-Object -First 1
```

The preceding command has the following output:

```

ScheduledTask      : VMware.Vim.ScheduledTaskEventArgument
Entity             : VMware.Vim.ManagedEntityEventArgument
Key                : 64835
ChainId            : 64835

    CreatedTime      : 1/7/2017 9:04:01 PM
UserName           :
Datacenter         :
ComputeResource    :
Host               :
Vm                 :
Ds                 :
Net                :
Dvs                :
FullFormattedMessage : Running task VMware vCenter Update Manager
                        Check Notification on Datacenters in
                        datacenter
ChangeTag          :

```

The output has a property `CreatedTime`. The value of the `CreatedTime` property has a `DateTime` object type.

Let's try to pipe the output of the preceding command into the `Get-Date` cmdlet:

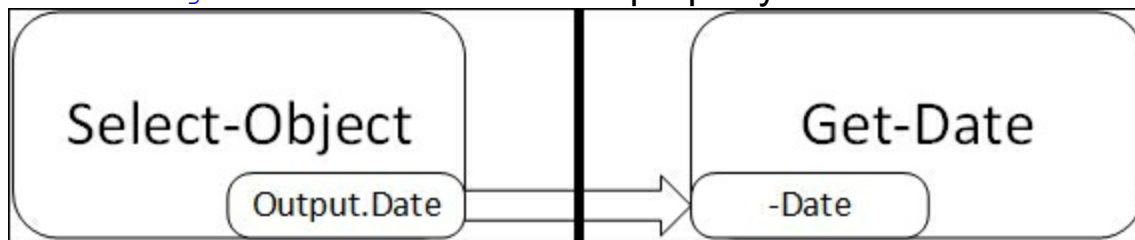
```
VMware PowerCLI 6.5 Release 1 build 4624819
PowerCLI C:\> Get-VIEvent | Select-Object -First 1 | Get-Date
Get-Date : The input object cannot be bound to any parameters for
the command either because the command does not take pipeline input
or the input and its properties do not match any of the parameters
that take pipeline input.
At line:1 char:40
+ Get-VIEvent | Select-Object -First 1 | Get-Date
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (VMware.Vim.UserLogo
utSessionEvent:PSObject) [Get-Date], ParameterBindingException
+ FullyQualifiedErrorId : InputObjectNotBound,Microsoft.PowerSh
ell.Commands.GetDateCommand
```

This gives an error message because the output of the `Get-VIEvent` cmdlet does not have a `Date` property that matches the `-Date` parameter of the `Get-Date` cmdlet. We will now use a calculated property to rename the `CreatedTime` property into `Date`:

```
PowerCLI C:\> Get-VIEvent | Select-Object -First 1 |
>> Select-Object @{Name="Date";Expression={$_.CreatedTime}} |
>> Get-Date
>>
Saturday, January 7, 2017 9:14:51 PM
```

Because the output of the `Select-Object` cmdlet has a `Date` property and this property matches the `Get-Date` parameter `-Date` using the `ByPropertyName` parameter binding, the pipeline now works.

The following figure shows that the pipeline binds the value of the `Date` property in the output of the `Select-Object` cmdlet to the `-Date` property of the `Get-Date` cmdlet:



Most PowerCLI cmdlets will accept input from the pipeline using the `ByValue` parameter binding. However, only a few PowerCLI cmdlets will accept input from the pipeline using the `ByPropertyName` parameter binding. You can find these cmdlets with the following PowerShell code:

```
PowerCLI C:\> Get-VICommand |
>> Where-Object {$_.ParameterSets.Parameters |
>> Where-Object {
>> $_.ValueFromPipeline -and $_.ValueFromPipelineByPropertyName}}
>>
```

The preceding command has the following output:

CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Add-EsxSoftwareDepot	6.0.0.0	VMware.ImageBuilder
Cmdlet	Add-EsxSoftwarePackage	6.0.0.0	VMware.ImageBuilder
Cmdlet	Compare-EsxImageProfile	6.0.0.0	VMware.ImageBuilder
Cmdlet	Export-EsxImageProfile	6.0.0.0	VMware.ImageBuilder
Cmdlet	New-EsxImageProfile	6.0.0.0	VMware.ImageBuilder
Cmdlet	Remove-EsxImageProfile	6.0.0.0	VMware.ImageBuilder
Cmdlet	Remove-EsxSoftwareDepot	6.0.0.0	VMware.ImageBuilder
Cmdlet	Remove-EsxSoftwarePackage	6.0.0.0	VMware.ImageBuilder
Cmdlet	Set-EsxImageProfile	6.0.0.0	VMware.ImageBuilder

The script uses the `Get-VICommand` cmdlet to get all of the PowerCLI cmdlets. It then filters only those cmdlets that have parameters that accept pipeline inputs using `ByPropertyName`. As you see, when writing this book, only cmdlets from the PowerCLI `VMware.ImageBuilder` module accept pipeline inputs using `ByPropertyName`.

Using the PowerShell object cmdlets

PowerShell has some cmdlets that are designed to work with all kinds of objects. You can easily recognize them because they all have the noun `Object`. You can use the `Get-Command` cmdlet `-Noun` parameter to find them:

```
PowerCLI C:\> Get-Command -Noun Object
```

The preceding command has the following output:

CommandType	Name	Version	Source
Cmdlet	Compare-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	ForEach-Object	3.0.0.0	Microsoft.PowerShell.Core
Cmdlet	Group-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Measure-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	New-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Select-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Sort-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Tee-Object	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Where-Object	3.0.0.0	Microsoft.PowerShell.Core

In the following section, we will discuss the `Object` cmdlets.

Using the Select-Object cmdlet

If you want to retrieve a subset of the properties of an object, select unique objects or a specific number of objects or specific objects from an array, you can use the `Select-Object` cmdlet. You can also use the `Select-Object` cmdlet to add properties to an object using calculated properties, as you have seen in [Chapter 2](#), Learning Basic PowerCLI Concepts.

The `Select-Object` cmdlet has the following syntax. The first parameter set is the default:

```
Select-Object [[-Property] <Object[]>] [-ExcludeProperty  
<String[]>] [-ExpandProperty <String>] [-First <Int32>] [-  
InputObject <PSObject>] [-Last <Int32>] [-Skip <Int32>] [-Unique]  
[-Wait] [<CommonParameters>]
```

The second parameter set can be used to skip the last part of the output:

```
Select-Object [[-Property] <Object[]>] [-ExcludeProperty  
<String[]>] [-ExpandProperty <String>] [-InputObject <PSObject>] [-  
SkipLast <Int32>] [-Unique] [<CommonParameters>]
```

The third parameter set can be used to specify an array of objects to return based on their index values:

```
Select-Object [-Index <Int32[]>] [-InputObject <PSObject>] [-  
Unique] [-Wait] [<CommonParameters>]
```

You can use `Select` as an alias for the `Select-Object` cmdlet.

The `Get-VM` cmdlet returns the `Name`, `PowerState`, `NumCpu`, and `MemoryGB` properties of the virtual machines by default. But what if you want to return the `Name`, `VMHost`, and `Cluster` properties instead? If you look at the properties of a virtual machine object, you will see the `VMHost` property. But you will not see a `Cluster` property. However, if you look at a `VMHostImpl` object, you will find a `Parent` property. The `Parent` property of a `VMHostImpl`

object contains the cluster that the host is a member of. Technet24.ir

Note

You can use this information to create a `PowerCLI` one-liner to get the host and cluster of all of the virtual machines:

```
PowerCLI C:\> Get-VM | Select-Object -Property
                Name,VMHost,
                @{Name="Cluster";Expression={$_.VMHost.Parent}}
Name VMHost      Cluster
----
DC1  192.168.0.133 Cluster01
```

The preceding command uses the `Select-Object` cmdlet to select the `Name` and `VMHost` properties of the virtual machine objects, and it creates a calculated property named `Cluster` that retrieves the cluster via the `VMHost.Parent` property.

Note

You can create a one-liner from all of your PowerShell scripts by using the semicolon as a separator between the commands. Use this only on the command line. It makes your scripts hard to read.

You can use the `Select-Object -First` parameter to specify the number of objects to select from the beginning of an array of input objects. For example, to retrieve the first three host types, use the following command:

```
PowerCLI C:\> Get-VMHost | Select-Object -First 3
```

If you are typing commands at the pipeline, you can also use their aliases. For `Select-Object`, the alias is `Select`. So, the next example will give the same result as the preceding one:

```
PowerCLI C:\> Get-VMHost | Select -First 3
```

To select a number of objects starting from the end of an array of objects, use the `Select-Object -Last` parameter. The following command retrieves the last cluster:

```
PowerCLI C:\> Get-Cluster | Select-Object -Last 1
```

You can also skip objects from the beginning or the end of an array using the `Select-Object -Skip` parameter. The following command returns all of the folder objects except the first two:

```
PowerCLI C:\> Get-Folder | Select-Object -Skip 2
```

Tip

A very interesting parameter of the `Select-Object` cmdlet is the `-ExpandProperty` parameter. You can use this parameter to expand the object if the property contains an object. For example, if you want to get the `VMHostImpl` object of the virtual machine named `dc1`, you can execute the following command:

```
PowerCLI C:\> Get-VM -Name dc1 | Select-Object
                -ExpandProperty VMHost
```

Using the Where-Object cmdlet

If you only want a subset of all of the objects that a command returns, you can use the `Where-Object` cmdlet to filter the output of a command and only return the objects that match the criteria of the filter.

The `Where-Object` cmdlet syntax definition is so long that it'll take too much space in this book. You can easily get the `Where-Object` cmdlet syntax with the following command:

```
PowerCLI C:\> Get-Help Where-Object
```

PowerShell V3 introduced a new, easier syntax for the `Where-Object` cmdlet. I will show you both the V2 and V3 syntaxes. First, let's see the new PowerShell V3 syntax. Let's try to find all virtual machines that have only one virtual CPU. You can do this by searching for virtual machines that have a `NumCpu` property with a value of `1`:

```
PowerCLI C:\> Get-VM | Where-Object NumCpu -eq 1
Name      PowerState  Num CPUs  MemoryGB
----      -
DC1       PoweredOff  1         0.250
```

If you use the alias `Where`, the command looks more like a natural language:

```
PowerCLI C:\> Get-VM | Where NumCpu -eq 1
```

You can also use the alias `?` if you want to type less on the command line:

```
PowerCLI C:\> Get-VM | ? NumCpu -eq 1
```

The **PowerShell V2** syntax is a bit more obscure. You have to use a script block as the value of the `Where-Object -FilterScript` parameter:

```
PowerCLI C:\> Get-VM | Where-Object -FilterScript {$_.NumCpu -eq 1}
```

Because the `-FilterScript` parameter is the first positional parameter of the `Where-Object` cmdlet, nobody uses the parameter name, and you will always see one of the following command lines being used:

```
PowerCLI C:\> Get-VM | Where-Object {$_.NumCpu -eq 1}
PowerCLI C:\> Get-VM | where {$_.NumCpu -eq 1}
```

The advantage of the PowerShell V2 syntax over the V3 syntax is that you can create complex filtering scripts. For example:

```
PowerCLI C:\> Get-VM |
>> Where-Object {$_.NumCpu -gt 2 -and $_.MemoryGB -lt 16}
>>
```

The preceding command will show you all of the virtual machines with more than two virtual CPUs and less than 16 GB of memory. If you want to create the same filter using the PowerShell V3 syntax, you have to use two filters: one for the number of CPUs and one for the memory:

```
PowerCLI C:\> Get-VM | Where NumCpu -gt 2 | Where MemoryGB -lt 16
```

Using the `ForEach-Object` cmdlet

Some cmdlets don't accept properties from the pipeline. On the other hand, you would like to use a cmdlet in the pipeline, but the property you want to use in the pipeline doesn't accept pipeline input. This is where the PowerShell `ForEach-Object` cmdlet will help you.