



Quick answers to common problems

Microsoft Exchange Server 2013 PowerShell Cookbook

Second Edition

Over 120 recipes to help manage and administrate Exchange
Server 2013 with PowerShell 3

Jonas Andersson
Mike Pfeiffer

[PACKT] enterprise 
PUBLISHING professional expertise distilled

Microsoft Exchange Server 2013 PowerShell Cookbook

Second Edition

Over 120 recipes to help manage and administrate
Exchange Server 2013 with PowerShell 3

Jonas Andersson

Mike Pfeiffer



BIRMINGHAM - MUMBAI

Microsoft Exchange Server 2013 PowerShell Cookbook

Second Edition

Copyright © 2013 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 authors, 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: July 2011

Second Edition: May 2013

Production Reference: 1100513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-942-7

www.packtpub.com

Cover Image by David Gimenez (bilbaorocker@yahoo.co.uk)

Credits

Authors

Jonas Andersson

Mike Pfeiffer

Reviewers

Marcelo Vighi Fernandes

Anderson Patricio

Acquisition Editor

Andrew Duckworth

Lead Technical Editor

Neeshma Ramakrishnan

Technical Editors

Dennis John

Dominic Pereira

Nitee Shetty

Project Coordinator

Arshad Sopariwala

Proofreaders

Maria Gould

Paul Hindle

Indexer

Hemangini Bari

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Authors

Jonas Andersson is a devoted person who is constantly developing himself and his skills. He started in the IT business in 2004 and worked at first in a support center where he got his basic knowledge. In 2007 he started his career as a Microsoft infrastructure consultant and from 2008 onwards his focus has been on Microsoft Exchange.

Even though his focus is on Microsoft Exchange, his interests include migrations, backup, storage, and archiving. At the start of 2010, he was employed at a large outsourcing company as a messaging specialist, specializing in Microsoft Exchange. His work includes designing, implementing, and developing messaging solutions for enterprise customers.

His unique knowledge makes him a key figure in large and complex migration projects where he works with design and implementation. Examples of these projects include migrations from the IBM Domino mail platform to Microsoft Exchange 2007/2010 and Office 365, using Quest Software with full coexistence between the systems for mail flow, directory synchronization, and free busy lookups.

Apart from his daily job, he was active on TechNet forums, he also writes articles at his blog (<http://www.testlabs.se/blog>), and Twitter and other social media.

As a reward for the work in the community he was been awarded the Microsoft Community Contributor Award both 2011 and 2012.

Acknowledgement

Since this is my first book, it's been a great experience and a great honor to get the opportunity to write an update of the great book that Mike Pfeiffer initially wrote for Microsoft Exchange 2010.

I look forward to continuing these kinds of side-projects to my regular work.

There are a lot of people I would like to thank; firstly of course my family, which includes my parents and my fiancée for the love and energy they keep on giving me. Besides my family I want to thank Magnus Björk and Mike Pfeiffer for answering my e-mails when I needed to verify things. I also want to thank Anderson Patricio and Marcelo Vighi for doing great work with the technical review and giving me lots of great feedback.

I hope that you will enjoy the book and that its content will help you to develop your skills in the area.

Mike Pfeiffer has been in the IT field for 15 years, and has been working on Exchange for the majority of that time. He is a Microsoft Certified Master and a former Microsoft Exchange MVP. These days he works at Microsoft as a Premier Field Engineer where he helps customers deploy and maintain Microsoft Exchange and Lync Server solutions. You can find his writings online at mikepfeiffer.net, where he occasionally blogs about Exchange, Lync, and PowerShell-related topics.

About the Reviewers

Marcelo Vighi Fernandes has over 14 years of experience in the IT field, always focusing on Microsoft Exchange Server, Active Directory, and other Microsoft Infrastructure solutions. Currently he is working at SolarWinds Inc. as a Technical Sales Engineer in Brazil. Marcelo is a well-known writer for a very important Exchange resource website in Portuguese where he and others members of the community add content on a weekly basis. You can reach this site at www.andersonpatricio.org.

He also contributes to many activities within the Exchange and cloud computing communities, such as presentations, articles, tutorials, and he also has two blogs on Exchange Server and cloud computing.

Anderson Patricio is an Exchange Server MVP and a messaging consultant based in Toronto, Canada, designing and deploying solutions for clients located in North and South America. He has been working with Exchange since Version 5 and he had the opportunity to use PowerShell since its beta release (code name Monad at the time).

He contributes to the Microsoft communities in several ways. In English, his blog, www.andersonpatricio.ca, is updated regularly with Exchange, PowerShell, and Microsoft as its general content. In Portuguese, he has an Exchange resource site (www.andersonpatricio.org) and he is also a TechEd presenter in South America and also creator of a couple of Exchange trainings in the Brazilian **Microsoft Virtual Academy (MVA)**.

You can also follow him on Twitter at <http://twitter.com/apatricio>.

He has also been the reviewer of several books such as *Windows PowerShell in Action*, Bruce Payette, Manning Publications; *PowerShell in Practice*, Richard Siddaway, Manning Publications; and *Microsoft Exchange 2010 PowerShell Cookbook*, Mike Pfeiffer, Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

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

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following [@PacktEnterprise](#) on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: PowerShell Key Concepts	7
Introduction	8
Using the help system	9
Understanding command syntax and parameters	13
Understanding the pipeline	17
Working with variables and objects	20
Formatting output	25
Working with arrays and hash tables	28
Looping through items	33
Creating and running scripts	35
Using flow control statements	39
Creating custom objects	43
Creating PowerShell functions	47
Setting up a profile	51
Chapter 2: Exchange Management Shell Common Tasks	55
Introduction	55
Manually configuring remote PowerShell connections	57
Using explicit credentials with PowerShell cmdlets	61
Transferring files through remote shell connections	62
Dealing with concurrent pipelines in remote PowerShell	65
Managing domains or an entire forest using recipient scope	67
Exporting reports to text and CSV files	68
Sending SMTP e-mails through PowerShell	72
Scheduling scripts to run at a later time	75
Logging shell sessions to a transcript	77
Automating tasks with the scripting agent	78
Scripting an Exchange server installation	81

Chapter 3: Managing Recipients	85
Introduction	86
Adding, modifying, and removing mailboxes	87
Working with contacts	91
Managing distribution groups	93
Managing resource mailboxes	95
Creating recipients in bulk using a CSV file	97
Working with recipient filters	101
Adding and removing recipient e-mail addresses	104
Hiding recipients from address lists	107
Configuring recipient moderation	108
Configuring message delivery restrictions	111
Managing automatic replies and out of office settings for a user	113
Adding, modifying, and removing server-side inbox rules	115
Managing mailbox folder permissions	118
Importing user photos into Active Directory	121
Chapter 4: Managing Mailboxes	125
Introduction	126
Reporting on the mailbox size	127
Working with move requests and performing mailbox moves	129
Mailbox move e-mail notification	134
Importing and exporting mailboxes	136
Deleting messages from mailboxes	141
Managing disconnected mailboxes	144
Generating mailbox folder reports	148
Reporting on mailbox creation time	151
Checking mailbox logon statistics	153
Setting storage quotas for mailboxes	154
Finding inactive mailboxes	155
Detecting and fixing corrupt mailboxes	157
Restoring deleted items from mailboxes	160
Managing public folder mailboxes	162
Reporting on public folder statistics	164
Managing user access to public folders	165
Chapter 5: Distribution Groups and Address Lists	169
Introduction	170
Reporting on distribution group membership	170
Adding members to a distribution group from an external file	172
Previewing dynamic distribution group membership	174
Excluding hidden recipients from a dynamic distribution group	176

Converting and upgrading distribution groups	179
Allowing managers to modify group membership	181
Removing disabled user accounts from distribution groups	183
Working with distribution group naming policies	185
Working with distribution group membership approval	187
Creating address lists	189
Exporting address list membership to a CSV file	191
Configuring hierarchical address books	193
Chapter 6: Mailbox Database Management	197
Introduction	197
Managing the mailbox databases	198
Moving databases and logs to another location	201
Configuring the mailbox database limits	205
Reporting on mailbox database size	207
Finding the total number of mailboxes in a database	209
Determining the average mailbox size per database	212
Reporting on database backup status	214
Restoring data from a recovery database	217
Chapter 7: Managing Client Access	221
Introduction	221
Managing ActiveSync, OWA, POP3, and IMAP4 mailbox settings	223
Setting internal and external CAS URLs	225
Managing Outlook Anywhere settings	229
Blocking Outlook clients from connecting to Exchange	231
Reporting on active OWA and RPC connections	234
Controlling ActiveSync device access	237
Reporting on ActiveSync devices	239
Chapter 8: Managing Transport Service	243
Introduction	243
Managing connectors	245
Configuring transport limits	248
Allowing application servers to relay mail	250
Managing transport rules and settings	253
Creating a basic disclaimer	260
Working with custom DSN messages	261
Managing connectivity and protocol logs	264
Searching message tracking logs	269
Working with messages in transport queues	273
Searching anti-spam agent logs	278
Implementing a header firewall	282

Chapter 9: High Availability	285
Introduction	285
Building a Windows NLB cluster for CAS servers	287
Creating a Database Availability Group	291
Adding mailbox servers to a Database Availability Group	293
Configuring Database Availability Group network settings	295
Adding mailbox copies to a Database Availability Group	297
Activating mailbox database copies	300
Working with lagged database copies	302
Reseeding a database copy	304
Using the automatic reseed feature	305
Performing maintenance on Database Availability Group members	309
Reporting on database status, redundancy, and replication	312
Chapter 10: Exchange Security	319
Introduction	319
Granting users full access permissions to mailboxes	320
Finding users with full access to mailboxes	323
Sending e-mail messages as another user or group	325
Working with Role Based Access Control (RBAC)	326
Creating a custom RBAC role for administrators	330
Creating a custom RBAC role for end users	332
Troubleshooting Role Based Access Control	336
Generating a certificate request	338
Installing certificates and enabling services	340
Importing certificates on multiple exchange servers	342
Chapter 11: Compliance and Audit Logging	347
Introduction	347
Managing archive mailboxes	349
Configuring archive mailbox quotas	350
Creating retention tags and policies	352
Applying retention policies to mailboxes	357
Placing mailboxes on retention hold	358
Placing mailboxes on in-place hold	360
Performing a discovery search	363
Enabling mailbox audit logging	366
Generating mailbox audit log reports	368
Configuring Administrator Audit Logging	371
Searching the administrator audit logs	373

Chapter 12: Server Monitoring and Troubleshooting	377
Introduction	378
Managing and monitoring services	379
Verifying server connectivity	383
Working with event logs	385
Reporting on disk usage	388
Checking CPU utilization	391
Monitoring memory utilization	395
Reporting on Exchange Server uptime	397
Troubleshooting the Mailbox role	400
Troubleshooting the Client Access Server role	402
Troubleshooting the Transport service	405
Verifying certificate health	406
Chapter 13: Scripting with the Exchange Web Services Managed API	411
Introduction	411
Getting connected to EWS	413
Sending e-mail messages with EWS	417
Working with impersonation	422
Searching mailboxes	427
Retrieving the headers of an e-mail message	432
Deleting e-mail items from a mailbox	437
Creating calendar items	442
Exporting attachments from a mailbox	447
Appendix A: Common Shell Information	453
Exchange Management Shell reference	453
Properties that can be used with the Filter parameter	459
Properties that can be used with the RecipientFilter parameter	462
Appendix B: Query Syntaxes	465
Advanced Query Syntax	465
Using the word phrase search	466
Using a date range search	468
Using the message type search	469
Using the logical connector search	470
Index	471

Preface

This book is full of immediately usable task-based recipes for managing and maintaining your Microsoft Exchange 2013 environment with Windows PowerShell 3.0 and the Exchange Management Shell. The focus of this book is to show you how to automate routine tasks and solve common problems. While the Exchange Management Shell literally provides hundreds of cmdlets, we will not cover every single one of them individually. Instead, we'll focus on the common, real world scenarios. You'll be able to leverage these recipes right away, allowing you to get the job done quickly, and the techniques that you'll learn will allow you to write your own amazing one-liners and scripts with ease.

What this book covers

Chapter 1, PowerShell Key Concepts, introduces several PowerShell core concepts such as command syntax and parameters, working with the pipeline, and flow control with loops and conditional logic. The topics covered in this chapter lay the foundation for the remaining code samples in each chapter.

Chapter 2, Exchange Management Shell Command Tasks, covers day-to-day tasks and general techniques for managing Exchange from the command line. The topics include configuring manual remote shell connections, exporting reports to external files, sending e-mail messages from scripts, and scheduling scripts to run with the Task Scheduler.

Chapter 3, Managing Recipients, demonstrates some of the most common recipient-related management tasks, such as creating mailboxes, distribution groups, and contacts. You'll also learn how to manage server-side inbox rules, out of office settings, and import user photos into the Active Directory.

Chapter 4, Managing Mailboxes, shows you how to perform various mailbox management tasks that include moving mailboxes, importing and exporting mailbox data, and the detection and repair of corrupt mailboxes. In addition, you'll learn how to delete and restore items from a mailbox, manage the new public folders, and generate some basic reports.

Chapter 5, Distribution Groups and Address Lists, takes you deeper into distribution group management. The topics include distribution group reporting, distribution group naming policies, and allowing end users to manage distribution group membership. You'll also learn how to create address lists and hierarchal address books.

Chapter 6, Mailbox Database Management, shows how to set database settings and limits. Report generation for mailbox database size, average mailbox size per database, and backup status is also covered in this chapter.

Chapter 7, Managing Client Access, covers the managing of ActiveSync, OWA, POP, and IMAP. It also covers the configuration of these components in Exchange 2013. We'll also take a look at controlling connections from various clients, including ActiveSync devices.

Chapter 8, Managing Transport Service, explains the various methods used to control mail flow within your Exchange organization. You'll learn how to create, send, and receive connectors, allow application servers to relay mail, and manage transport queues.

Chapter 9, High Availability, covers the implementation and management tasks related to Database Availability Groups (DAGs). Topics include creating DAGs, adding mailbox database copies, and performing maintenance on DAG members. It also covers the new feature called automatic reseed.

Chapter 10, Exchange Security, introduces the new Role Based Access Control (RBAC) permissions model. You'll learn how to create custom RBAC roles for administrators and end users, and also how to manage mailbox permissions and implement SSL certificates.

Chapter 11, Compliance and Audit Logging, covers the new compliance and auditing features included in Exchange 2013. Topics such as archiving mailboxes and discovery search are covered here, as well as administrator and mailbox audit logging.

Chapter 12, Server Monitoring and Troubleshooting, shows you how to monitor and report on service availability and resource utilization using PowerShell core cmdlets and WMI. Event log monitoring and Exchange server role troubleshooting tactics are also covered.

Chapter 13, Scripting with the Exchange Web Services Managed API, introduces advanced scripting topics that leverage Exchange Web Services. In this chapter, you'll learn how to write scripts and functions that go beyond the capabilities of the Exchange Management Shell cmdlets.

Appendix A, Common Shell Information, is a reference for the variables, scripts, and the filtering functions. These references will help you when writing scripts or running interactive.

Appendix B, Query Syntaxes, is a reference for the Advanced Query Syntax (AQS). Here are lots of different examples that can be used in the real world.

What you need for this book

To complete the recipes in this book, you'll need the following:

- ▶ PowerShell v3, which is already installed by default on Windows 8 and Windows Server 2012.
- ▶ A fully operational lab environment with an Active Directory forest and Exchange organization.
- ▶ Ideally, your Exchange Servers will run Windows Server 2012, but they can run Windows Server 2008 R2, if needed.
- ▶ You'll need to have at least one Microsoft Exchange 2013 server.
- ▶ It is assumed that the account you are using is a member of the Organization Management role group. The user account used to install Exchange 2013 is automatically added to this group.
- ▶ If possible, you'll want to run the commands, scripts, and functions in this book from a client machine. The 64-bit version of Windows 8 with the Exchange 2013 Management Tools installed is a good choice. You can also run the tools on Windows 7. Each client will need some additional prerequisites in order to run the tools, see Microsoft's TechNet documentation for full details.
- ▶ If you don't have a client machine, you can run the Exchange Management Shell from an Exchange 2013 server.
- ▶ *Chapter 13* requires the Exchange Web Services Managed API version 2.0, which can be downloaded from <http://www.microsoft.com/en-us/download/details.aspx?id=35371>.

The code samples in this book should be run in a lab environment and fully tested before deployed into production. If you don't have a lab environment setup, you can download the software from <http://technet.microsoft.com/en-us/exchange/>. Then build the servers on your preferred virtualization engine.

Who this book is for

This book is for messaging professionals who want to learn how to build real-world scripts with Windows PowerShell 3.0 and the Exchange Management Shell. If you are a network or systems administrator responsible for managing and maintaining the on-premise version of Exchange Server 2013, then this book is for you.

The recipes in this Cookbook touch on each of the core Exchange 2013 server roles, and require a working knowledge of the supporting technologies, such as Windows Server 2008, 2008 R2 or 2012, Active Directory, and DNS.

All of the topics in this book are focused on the on-premises version of Exchange 2013, and we will not cover Microsoft's hosted version of Exchange Online through Office 365. However, the concepts you'll learn in this book will allow you to hit the ground running with that platform since it will give you an understanding of PowerShell's command syntax and object-based nature.

Conventions

In this book, you will find a number of styles of text 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: "We can read the content of an external file into the shell using the `Get-Content` cmdlet"

Commands and blocks of code are set as follows:

```
Get-Mailbox -ResultSize Unlimited | Out-File C:\report.txt
```

Commands like this can be invoked interactively in the shell, or from within a script or function.

Most of the commands you'll be working with will be very long. In order for them to fit into the pages of this book, we'll need to use line continuation. For example, here is a command that creates a mailbox:

```
New-Mailbox -UserPrincipalName jsmith@contoso.com `
-FirstName John `
-LastName Smith `
-Alias jsmith `
-Database DB1 `
-Password $password
```

Notice that the last character on each line is the backtick (```) symbol, also referred to as the grave accent. This is PowerShell's line continuation character. You can run this command as is, but make sure there aren't any trailing spaces at the end of each line. You can also remove the backticks and carriage returns and run the command on one line. Just ensure the spaces between the parameters and arguments are maintained.

You'll also see long pipeline commands formatted like the following example:

```
Get-Mailbox -ResultSize Unlimited |
Select-Object DisplayName,ServerName,Database |
Export-Csv c:\mbreport.csv -NoTypeInformation
```

PowerShell uses the pipe character (|) to send object output from a command down the pipeline so it can be used as input by another command. The pipe character does not need to be escaped. You can enter the previous command as is, or you can format the command so that everything is on one line.

Any command-line input or output that must be done interactively at the shell console is written as follows:

```
[PS] C:\>Get-Mailbox administrator | ft ServerName,Database -Auto
```

```
ServerName Database
```

```
-----
```

```
mbx1          DB01
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Exchange Management Shell** shortcut".



Warnings or important notes appear in a box like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

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 on 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 all Packt books you have purchased 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.

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 would 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/support>, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright 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

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

PowerShell Key Concepts

In this chapter, we will cover the following:

- ▶ Using the help system
- ▶ Understanding command syntax and parameters
- ▶ Understanding the pipeline
- ▶ Working with variables and objects
- ▶ Formatting output
- ▶ Working with arrays and hash tables
- ▶ Looping through items
- ▶ Creating and running scripts
- ▶ Using flow control statements
- ▶ Creating custom objects
- ▶ Creating PowerShell functions
- ▶ Setting up a profile

Introduction

So, your organization has decided to move to Exchange Server 2013 to take advantage of the many exciting new features such as integrated e-mail archiving, discovery capabilities, and high availability functionality. Like it or not, you've realized that PowerShell is now an integral part of Exchange Server management and you need to learn the basics to have a point of reference for building your own scripts. That's what this book is all about. In this chapter, we'll cover some core PowerShell concepts that will provide you with a foundation of knowledge for using the remaining examples in this book. If you are already familiar with PowerShell, you may want to use this chapter as a review or as a reference for later on after you've started writing scripts.

If you're completely new to PowerShell, its concept may be familiar if you've worked with Unix command shells. Like Unix-based shells, PowerShell allows you to string multiple commands together on one line using a technique called pipelining. This means that the output of one command becomes the input for another. But, unlike Unix shells that pass text output from one command to another, PowerShell uses an object model based on the .NET Framework, and objects are passed between commands in a pipeline, as opposed to plain text. From an Exchange perspective, working with objects gives us the ability to access very detailed information about servers, mailboxes, databases, and more. For example, every mailbox you manage within the shell is an object with multiple properties, such as an e-mail address, database location, or send and receive limits. The ability to access this type of information through simple commands means that we can build powerful scripts that generate reports, make configuration changes, and perform maintenance tasks with ease.

Performing some basic steps

To work with the code samples in this chapter, follow these steps to launch the Exchange Management Shell:

1. Log on to a workstation or server with the Exchange Management Tools installed.
2. You can connect using remote PowerShell if you for some reason don't have Exchange Management Tools installed. Use the following command:

```
$Session = New-PSSession -ConfigurationName Microsoft.Exchange `
-ConnectionUri http://tlex01/PowerShell/ `
-Authentication Kerberos `
Import-PSSession $Session
```
3. Open the **Exchange Management Shell** by clicking on **Start | All Programs | Microsoft Exchange Server 2013**. Or if you're using Windows 2012 Server, it can be found by pressing the Windows key.
4. Click on the **Exchange Management Shell** shortcut.



Remember to start the Exchange Management Shell using **Run As Admin** to avoid permission problems.

In the chapter, notice that in the examples of cmdlets, I have used the back tick (``) character for breaking up long commands into multiple lines. The purpose with this is to make it easier to read. The back ticks are not required and should only be used if needed.

Using the help system

The Exchange Management Shell includes over 750 cmdlets (pronounced command-lets), each with a set of multiple parameters. For instance, the `New-Mailbox` cmdlet accepts more than 60 parameters, and the `Set-Mailbox` cmdlet has over 160 available parameters. It's safe to say that even the most experienced PowerShell expert would be at a disadvantage without a good help system. In this recipe, we'll take a look at how to get help in the Exchange Management Shell.

How to do it...

To get help information for a cmdlet, type `Get-Help`, followed by the cmdlet name. For example, to get help information about the `Get-Mailbox` cmdlet, run the following command:

```
Get-Help Get-Mailbox -full
```



Downloading the example code

You can download the example code files for all Packt books you have purchased 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.

How it works...

When running `Get-Help` for a cmdlet, a synopsis and description for the cmdlet will be displayed in the shell. The `Get-Help` cmdlet is one of the best discovery tools to use in PowerShell. You can use it when you're not quite sure how a cmdlet works or what parameters it provides.

You can use the following switch parameters to get specific information using the `Get-Help` cmdlet:

- **Detailed:** The detailed view provides parameter descriptions and examples, and uses the following syntax:

```
Get-Help<cmdletname>-Detailed
```

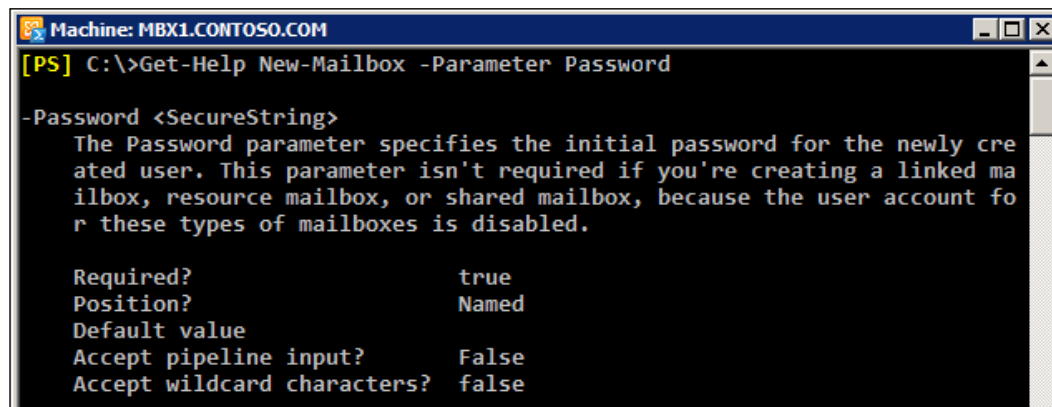

- ▶ **Examples:** You can view multiple examples of how to use a cmdlet by running the following syntax:

```
Get-Help<cmdletname>-Examples
```

- ▶ **Full:** Use the following syntax to view the complete contents of the help file for a cmdlet:

```
Get-Help<cmdletname>-Full
```

Some parameters accept simple strings as input, while others require an actual object. When creating a mailbox using the `New-Mailbox` cmdlet, you'll need to provide a secure string object for the `-Password` parameter. You can determine the data type required for a parameter using `Get-Help`:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Help New-Mailbox -Parameter Password

-Password <SecureString>
    The Password parameter specifies the initial password for the newly cre
    ated user. This parameter isn't required if you're creating a linked ma
    ilbox, resource mailbox, or shared mailbox, because the user account fo
    r these types of mailboxes is disabled.

    Required?                true
    Position?                Named
    Default value
    Accept pipeline input?   False
    Accept wildcard characters? false
```

You can see from the command output that we get several pieces of key information about the `-Password` parameter. In addition to the required data type of `<SecureString>`, we can see that this is a named parameter. It is required when running the `New-Mailbox` cmdlet and it does not accept wildcard characters. You can use `Get-Help` when examining the parameters for any cmdlet to determine whether or not they support these settings.

You could run `Get-HelpNew-MailboxExamples` to determine the syntax required to create a secure string password object and how to use it to create a mailbox. This is also covered in detail in the recipe entitled *Adding, modifying, and removing mailboxes* in *Chapter 3, Managing Recipients*.

There's more...

There will be times when you'll need to search for a cmdlet without knowing its full name. In this case, there are a couple of commands you can use to find the cmdlets you are looking for.

To find all cmdlets that contain the word "mailbox", you can use a wildcard, as shown in the following command:

```
Get-Command *Mailbox*
```

You can use the `-Verb` parameter to find all cmdlets starting with a particular verb:

```
Get-Command -Verb Set
```

To search for commands that use a particular noun, specify the name with the `-Noun` parameter:

```
Get-Command -Noun Mailbox
```

The `Get-Command` cmdlet is a built-in PowerShell core cmdlet, and it will return commands from both Windows PowerShell as well as the Exchange Management Shell. The Exchange Management Shell also adds a special function called `Get-Ex` command that will return only Exchange-specific commands.

In addition to getting cmdlet help for cmdlets, you can use `Get-Help` to view supplementary help files that explain general PowerShell concepts that focus primarily on scripting. To display the help file for a particular concept, type `Get-Help about_` followed by the concept name. For example, to view the help for the core PowerShell commands, type the following:

```
Get-Help about_Core_Commands
```

You can view the entire list of conceptual help files using the following command:

```
Get-Help about_*
```

Don't worry about trying to memorize all the Exchange or PowerShell cmdlet names. As long as you can remember `Get-Command` and `Get-Help`, you can search for commands and figure out the syntax to do just about anything.

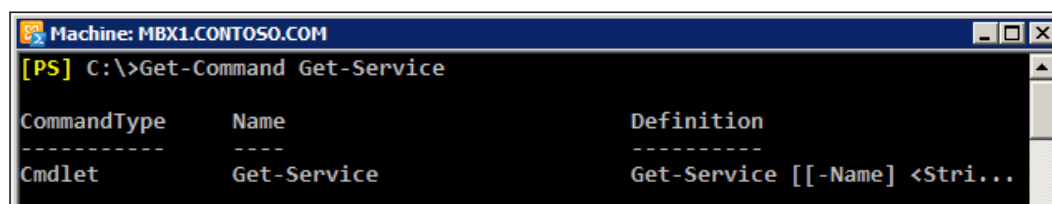
Getting help with cmdlets and functions

One of the things that can be confusing at first is the distinction between cmdlets and functions. When you launch the Exchange Management Shell, a remote PowerShell session is initiated to an Exchange server and specific commands, called proxy functions, are imported into your shell session. These proxy functions are essentially just blocks of code that have a name, such as `GetMailbox`, and that correspond to the compiled cmdlets installed on the server. This is true even if you have a single server and when you are running the shell locally on a server.

When you run the `Get-Mailbox` function from the shell, data is passed between your machine and the Exchange server through a remote PowerShell session. The `Get-Mailbox` cmdlet is actually executing on the remote Exchange server, and the results are being passed back to your machine. One of the benefits of this is that it allows you to run the cmdlets remotely regardless of whether your servers are on-premise or in the cloud. Additionally, this core change in the tool set is what allows Exchange 2010 and 2013 to implement its new security model by allowing and restricting which cmdlets administrators and end users can actually use through the shell or the web-based control panel.

We'll get into the details of all this throughout the remaining chapters in the book. The bottom line is that, for now, you need to understand that, when you are working with the help system, the Exchange 2013 cmdlets will show up as functions and not as cmdlets.

Consider the following command and the output:

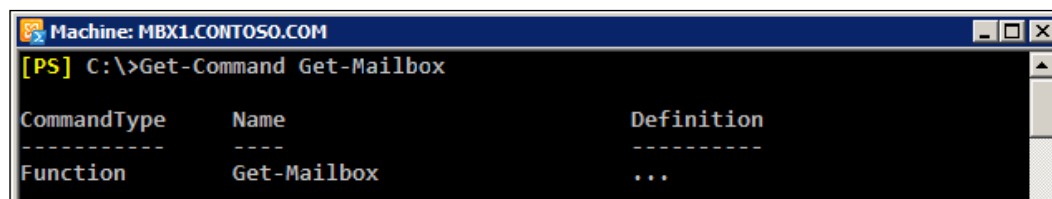


```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Command Get-Service

CommandType      Name                Definition
-----
Cmdlet            Get-Service         Get-Service [[-Name] <Stri...
```

Here we are running `Get-Command` against a PowerShell v3 core cmdlet. Notice that the `CmdletType` shows that this is a `Cmdlet`.

Now try the same thing for the `Get-Mailbox` cmdlet:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Command Get-Mailbox

CommandType      Name                Definition
-----
Function          Get-Mailbox         ...
```

As you can see, the `CommandType` for the `Get-Mailbox` cmdlet shows that it is actually a `Function`. So, there are a couple of key points to take away from this. First, throughout the course of this book, we will refer to the Exchange 2013 cmdlets as cmdlets, even though they will show up as functions when running `Get-Command`. Second, keep in mind that you can run `Get-Help` against any function name, such as `Get-Mailbox`, and you'll still get the help file for that cmdlet. But if you are unsure of the exact name of a cmdlet, use `Get-Command` to perform a wildcard search as an aid in the discovery process. Once you've determined the name of the cmdlet you are looking for, you can run `Get-Help` against that cmdlet for complete details on how to use it.

Try using the help system before going to the Internet to find answers. You'll find that the answers to most of your questions are already documented within the built-in cmdlet help.

See also

- ▶ The *Understanding command syntax and parameters* recipe
- ▶ The *Manually configuring remote PowerShell connections* recipe in *Chapter 2, Exchange Management Shell Common Tasks*
- ▶ The *Working with Role Based Access Control (RBAC)* recipe in *Chapter 10, Exchange Security*

Understanding command syntax and parameters

Windows PowerShell provides a large number of built-in cmdlets that perform specific operations. The Exchange Management Shell adds an additional set of PowerShell cmdlets used specifically for managing Exchange. We can also run these cmdlets interactively in the shell, or through automated scripts. When executing a cmdlet, parameters can be used to provide information, such as which mailbox or server to work with, or which attribute of those objects should be modified. In this recipe, we'll take a look at basic PowerShell command syntax and how parameters are used with cmdlets.

How to do it...

When running a PowerShell command, you type the cmdlet name, followed by any parameters required. Parameter names are preceded by a hyphen (-) followed by the value of the parameter. Let's start with a basic example. To get mailbox information for a user named `testuser`, use the following command syntax:

```
Get-Mailbox -Identity testuser
```

Alternatively, the following syntax also works and provides the same output, because the `-Identity` parameter is a positional parameter:

```
Get-Mailbox testuser
```

Most cmdlets support a number of parameters that can be used within a single command. We can use the following command to modify two separate settings on the `testuser` mailbox:

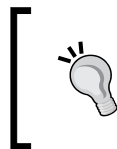
```
Set-Mailbox testuser -MaxSendSize 5mb -MaxReceiveSize 5mb
```

How it works...

All cmdlets follow a standard verb-noun naming convention. For example, to get a list of mailboxes you use the `Get-Mailbox` cmdlet. You can change the configuration of a mailbox using the `Set-Mailbox` cmdlet. In both examples, the verb (`Get` or `Set`) is the action you want to take on the noun (`Mailbox`). The verb is always separated from the noun using the hyphen (-) character. With the exception of a few Exchange Management Shell cmdlets, the noun is always singular.

Cmdlet names and parameters are not case sensitive. You can use a combination of upper and lowercase letters to improve the readability of your scripts, but it is not required.

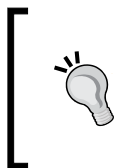
Parameter input is either optional or required, depending on the parameter and cmdlet you are working with. You don't have to assign a value to the `c` parameter since it is not required when running the `Get-Mailbox` cmdlet. If you simply run `Get-Mailbox` without any arguments, the first 1,000 mailboxes in the organization will be returned.



If you are working in a large environment with more than 1,000 mailboxes, you can run the `Get-Mailbox` cmdlet setting the `-ResultSize` parameter to `Unlimited` to retrieve all of the mailboxes in your organization.

Notice that in the first two examples we ran `Get-Mailbox` for a single user. In the first example, we used the `-Identity` parameter, but in the second example we did not. The reason we don't need to explicitly use the `-Identity` parameter in the second example is because it is a positional parameter. In this case, `-Identity` is in position 1, so the first argument received by the cmdlet is automatically bound to this parameter. There can be a number of positional parameters supported by a cmdlet, and they are numbered starting from one. Other parameters that are not positional are known as named parameters, meaning we need to use the parameter name to provide input for the value.

The `-Identity` parameter is included with most of the Exchange Management Shell cmdlets, and it allows you to classify the object you want to take an action on.



The `-Identity` parameter used with the Exchange Management Shell cmdlets can accept different value types. In addition to the alias, the following values can be used: `ADObjectID`, `Distinguished name`, `Domain\Username`, `GUID`, `LegacyExchangeDN`, `SmtpAddress`, and `User principal name (UPN)`.

Unlike the `Get-Mailbox` cmdlet, the `-Identity` parameter is required when you are modifying objects, and we saw an example of this when running the `Set-Mailbox` cmdlet. This is because the cmdlet needs to know which mailbox it should modify when the command is executed. When you run a cmdlet without providing input for a required parameter, you will be prompted to enter the information before execution.



In order to determine whether a parameter is required, named, or positional, supports wildcards, or accepts input from the pipeline, you can use the `Get-Help` cmdlet which is covered in the next recipe in this chapter.

Multiple data types are used for input depending on the parameter you are working with. Some parameters accept string values, while others accept integers or Boolean values. Boolean parameters are used when you need to set a parameter value to either true or false. PowerShell provides built-in shell variables for each of these values using the `$true` and `$false` automatic variables.



For a complete list of PowerShell v3 automatic variables, run `Get-Help about_automatic_variables`. Also see *Appendix A, Common Shell Information*, for a list of automatic variables added by the Exchange Management Shell.

For example, you can enable or disable a send connector using the `Set-SendConnector` cmdlet with the `-Enabled` parameter:

```
Set-SendConnector Internet -Enabled $false
```

Switch parameters don't require a value. Instead they are used to turn something on or off, or to either enable or disable a feature or setting. One common example of when you might use a switch parameter is when creating an archive mailbox for a user:

```
Enable-Mailbox testuser -Archive
```

PowerShell also provides a set of common parameters that can be used with every cmdlet. Some of the common parameters, such as the risk mitigation parameters (`-Confirm` and `-Whatif`), only work with cmdlets that make changes.

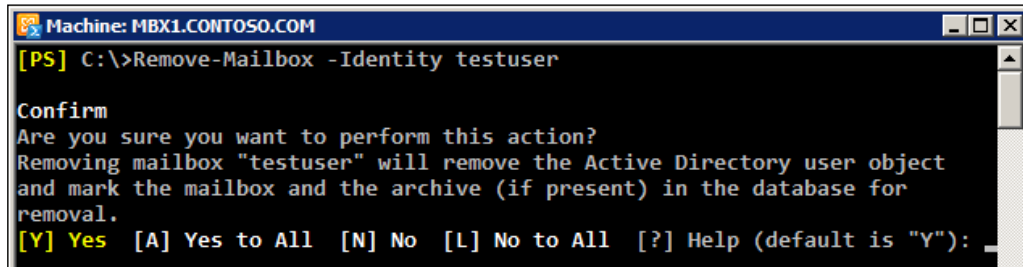


For a complete list of common parameters, run `Get-Help about_CommonParameters`.

Risk mitigation parameters allow you to preview a change or confirm a change that may be destructive. If you want to see what will happen when executing a command without actually executing it, use the `-WhatIf` parameter:

```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Mailbox -Database DB1 | Remove-Mailbox -WhatIf
What if: Removing mailbox "uss.local/Users/Test User" will remove the Active
Directory user object and mark the mailbox and the archive (if present) in
the database for removal.
[PS] C:\>_
```

When making a change, such as removing a mailbox, you'll be prompted for confirmation, as shown in the following screenshot:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Remove-Mailbox -Identity testuser

Confirm
Are you sure you want to perform this action?
Removing mailbox "testuser" will remove the Active Directory user object
and mark the mailbox and the archive (if present) in the database for
removal.
[Y] Yes [A] Yes to All [N] No [L] No to All [?] Help (default is "Y"):
```

To suppress this confirmation set the `-Confirm` parameter to false:

```
Remove-Mailbox testuser -Confirm:$false
```

Notice here that when assigning the `$false` variable to the `-Confirm` parameter, we had to use a colon immediately after the parameter name and then the Boolean value. This is different to how we assigned this value earlier with the `-Enabled` parameter when using the `Set-SendConnector` cmdlet. Remember that the `-Confirm` parameter always requires this special syntax, and while most parameters that accept a Boolean value generally do not require this, it depends on the cmdlet with which you are working. Fortunately, PowerShell has a great built-in help system that we can use when we run into these inconsistencies. When in doubt, use the help system, which is covered in detail in the next recipe.

Cmdlets and parameters support tab completion. You can start typing the first few characters of a cmdlet or a parameter name and hit the tab key to automatically complete the name or tab through a list of available names. This is very helpful in terms of discovery and can serve as a bit of a time saver.

In addition, you only need to type enough characters of a parameter name to differentiate it from another parameter name. The following command using a partial parameter name is completely valid:

```
Set-Mailbox -id testuser -Office Sales
```

Here we've used `id` as a shortcut for the `-Identity` parameter. The cmdlet does not provide any other parameters that start with `id`, so it automatically assumes you want to use the `-Identity` parameter.

Another helpful feature that some parameters support is the use of wildcards. When running the `Get-Mailbox` cmdlet, the `-Identity` parameter can be used with wildcards to return multiple mailboxes that match a certain pattern:

```
Get-Mailbox -id t*
```

In this example, all mailboxes starting with the letter "t" will be returned. Although this is fairly straightforward, you can refer to the help system for details on using wildcard characters in PowerShell by running `Get-Help about_Wildcards`.

There's more...

Parameter values containing a space need to be enclosed in either single or double quotation marks. The following command would retrieve all of the mailboxes in the Sales Users OU in Active Directory. Notice that since the OU name contains a space, it is enclosed in single quotes:

```
Get-Mailbox -OrganizationalUnit 'contoso.com/Sales Users/Phoenix'
```

Use double quotes when you need to expand a variable within a string:

```
$City = 'Phoenix'
```

```
Get-Mailbox -OrganizationalUnit "contoso.com/Sales Users/$City"
```

You can see here that we first create a variable containing the name of the city, which represents a sub OU under `Sales Users`. Next, we include the variable inside the string used for the organizational unit when running the `Get-Mailbox` cmdlet. PowerShell automatically expands the variable name inside the double quoted string where the value should appear and all mailboxes inside the `Phoenix` OU are returned by the command.



Quoting rules are documented in detail in the PowerShell help system. Run `Get-Helpabout_Quoting_Rules` for more information.

See also

- ▶ The *Using the help system* recipe
- ▶ The *Working with variables and objects* recipe

Understanding the pipeline

The single most important concept in PowerShell is the use of its flexible, object-based pipeline. You may have used pipelines in Unix-based shells, or when working with the `cmd.exe` command prompt. The concept of pipelines is similar to that of sending the output from one command to another. But, instead of passing plain text, PowerShell works with objects, and we can accomplish some very complex tasks in just a single line of code. In this recipe, you'll learn how to use pipelines to string together multiple commands and build powerful one-liners.

How to do it...

The following pipeline command would set the office location for every mailbox in the DB1 database:

```
Get-Mailbox -Database DB1 | Set-Mailbox -Office Headquarters
```

How it works...

In a pipeline, you separate a series of commands using the pipe (|) character. In the previous example, the `Get-Mailbox` cmdlet returns a collection of mailbox objects. Each mailbox object contains several properties that contain information such as the name of the mailbox, the location of the associated user account in Active Directory, and more. The `Set-Mailbox` cmdlet is designed to accept input from the `Get-Mailbox` cmdlet in a pipeline, and with one simple command we can pass along an entire collection of mailboxes that can be modified in one operation.

You can also pipe output to filtering commands, such as the `Where-Object` cmdlet. In this example, the command retrieves only the mailboxes with a `MaxSendSize` equal to 10 megabytes:

```
Get-Mailbox | Where-Object{$_ .MaxSendSize -eq 10mb}
```

The code that the `Where-Object` cmdlet uses to perform the filtering is enclosed in curly braces ({}). This is called a script block, and the code within this script block is evaluated for each object that comes across the pipeline. If the result of the expression is evaluated as true, the object is returned; otherwise, it is ignored. In this example, we access the `MaxSendSize` property of each mailbox using the `$_` object, which is an automatic variable that refers to the current object in the pipeline. We use the equals (`-eq`) comparison operator to check that the `MaxSendSize` property of each mailbox is equal to 10 megabytes. If so, only those mailboxes are returned by the command.



Comparison operators allow you to compare results and find values that match a pattern. For a complete list of comparison operators, run `Get-Helpabout_Comparison_Operators`.

When running this command, which can also be referred to as a one-liner, each mailbox object is processed one at a time using stream processing. This means that as soon as a match is found, the mailbox information is displayed on the screen. Without this behavior, you would have to wait for every mailbox to be found before seeing any results. This may not matter if you are working in a very small environment, but without this functionality in a large organization with tens of thousands of mailboxes, you would have to wait a long time for the entire result set to be collected and returned.

One other interesting thing to note about the comparison being done inside our `Where-Object` filter is the use of the `mb` multiplier suffix. PowerShell natively supports these multipliers and they make it a lot easier for us to work with large numbers. In this example, we've used `10mb`, which is the equivalent of entering the value in bytes because behind the scenes, PowerShell is doing the math for us by replacing this value with `1024*1024*10`. PowerShell provides support for the following multipliers: `kb`, `mb`, `gb`, `tb`, and `pb`.

There's more...

You can use advanced pipelining techniques to send objects across the pipeline to other cmdlets that do not support direct pipeline input. For example, the following one-liner adds a list of users to a group:

```
Get-User |
    Where-Object{$_ .title -eq "Exchange Admin"} | ForEach-Object{
        Add-RoleGroupMember -Identity "Organization Management" `
            -Member $_.name
    }
```

This pipeline command starts off with a simple filter that returns only the users that have their `title` set to `"Exchange Admin"`. The output from that command is then piped to the `ForEach-Object` cmdlet that processes each object in the collection. Similar to the `Where-Object` cmdlet, the `ForEach-Object` cmdlet processes each item from the pipeline using a script block. Instead of filtering, this time we are running a command for each user object returned in the collection and adding them to the `"Organization Management"` role group.

Using aliases in pipelines can be helpful because it reduces the number of characters you need to type. Take a look at the following command where the previous command is modified to use aliases:

```
Get-User |
    ?{$_ .title -eq "Exchange Admin"} | %{
        Add-RoleGroupMember -Identity "Organization Management" `
            -Member $_.name
    }
```

Notice the use of the question mark (`?`) and the percent sign (`%`) characters. The `?` character is an alias for the `Where-Object` cmdlet, and the `%` character is an alias for the `ForEach-Object` cmdlet. These cmdlets are used heavily, and you'll often see them used with these aliases because it makes the commands easier to type.



You can use the `Get-Alias` cmdlet to find all of the aliases currently defined in your shell session and the `New-Alias` cmdlet to create custom aliases.

The `Where-Object` and `ForEach-Object` cmdlets have additional aliases. Here's another way you could run the previous command:

```
Get-User |  
  where{$_ .title -eq "Exchange Admin"} | foreach{  
    Add-RoleGroupMember -Identity "Organization Management" `~  
    -Member $_.name  
  }
```

Use aliases when you're working interactively in the shell to speed up your work and keep your commands concise. You may want to consider using the full cmdlet names in production scripts to avoid confusing others who may read your code.

See also

- ▶ *The Looping through items recipe*
- ▶ *The Creating custom objects recipe*
- ▶ *The Dealing with concurrent pipelines in remote PowerShell recipe in Chapter 2, Exchange Management Shell Common Tasks*

Working with variables and objects

Every scripting language makes use of variables as placeholders for data, and PowerShell is no exception. You'll need to work with variables often to save temporary data to an object so you can work with it later. PowerShell is very different from other command shells in that everything you touch is, in fact, a rich object with properties and methods. In PowerShell, a variable is simply an instance of an object just like everything else. The properties of an object contain various bits of information depending on the type of object you're working with. In this recipe we'll learn to create user-defined variables and work with objects in the Exchange Management Shell.

How to do it...

To create a variable that stores an instance of the `testuser` mailbox, use the following command:

```
$mailbox = Get-Mailbox testuser
```

How it works...

To create a variable, or an instance of an object, you prefix the variable name with the dollar sign (\$). To the right of the variable name, use the equals (=) assignment operator, followed by the value or object that should be assigned to the variable. Keep in mind that the variables you create are only available during your current shell session and will be destroyed when you close the shell.

Let's look at another example. To create a string variable that contains an e-mail address, use the following command:

```
$email = "testuser@contoso.com"
```



In addition to user-defined variables, PowerShell also includes automatic and preference variables. To learn more, run `Get-Helpabout_Automatic_Variables` and `Get-Helpabout_Preference_Variables`.

Even a simple string variable is an object with properties and methods. For instance, every string has a `Length` property that will return the number of characters that are in the string:

```
[PS] C:\>$email.Length  
20
```

When accessing the properties of an object, you can use dot notation to reference the property with which you want to work. This is done by typing the object name, then a period, followed by the property name, as shown in the previous example. You access methods in the same way, except that the method names always end with parenthesis ().

The string data type supports several methods, such as `Substring`, `Replace`, and `Split`. The following example shows how the `Split` method can be used to split a string:

```
[PS] C:\>$email.Split("@")  
testuser  
contoso.com
```

You can see here that the `Split` method uses the "@" portion of the string as a delimiter and returns two substrings as a result.



PowerShell also provides a `-Split` operator that can split a string into one or more substrings. Run `Get-Helpabout_Split` for details.

There's more...

At this point, you know how to access the properties and methods of an object, but you need to be able to discover and work with these members. To determine which properties and methods are accessible on a given object, you can use the `Get-Member` cmdlet, which is one of the key discovery tools in PowerShell along with `Get-Help` and `Get-Command`.

To retrieve the members of an object, pipe the object to the `Get-Member` cmdlet. The following command will retrieve all of the instance members of the `$mailbox` object we created earlier:

```
$mailbox | Get-Member
```



To filter the results returned by `Get-Member`, use the `-MemberType` parameter to specify whether the type should be a `Property` or a `Method`.

Let's take a look at a practical example of how we could use `Get-Member` to discover the methods of an object. Imagine that each mailbox in our environment has had a custom `MaxSendSize` restriction set and we need to record the value for reporting purposes. When accessing the `MaxSendSize` property, the following information is returned:

```
[PS] C:\>$mailbox.MaxSendSize
IsUnlimited Value
-----
False          10 MB (10,485,760 bytes)
```

We can see here that the `MaxSendSize` property actually contains an object with two properties: `IsUnlimited` and `Value`. Based on what we've learned, we should be able to access the information for the `Value` property using dot notation:

```
[PS] C:\>$mailbox.MaxSendSize.Value
10 MB (10,485,760 bytes)
```

That works, but the information returned contains not only the value in megabytes, but also the total bytes for the `MaxSendSize` value. For the purpose of what we are trying to accomplish, we only need the total megabytes. Let's see if this object provides any methods that can help us out with this using `Get-Member`:

```
Machine: MBX1.CONTOSO.COM
[PS] C:\>$mailbox.MaxSendSize.Value | Get-Member -MemberType Method

TypeName: Microsoft.Exchange.Data.ByteQuantifiedSize

Name      MemberType Definition
-----
CompareTo Method    int CompareTo(Microsoft.Exchange.Data.ByteQuant...
Equals     Method    bool Equals(System.Object obj), bool Equals(Mic...
GetHashCode Method    int GetHashCode()
GetType    Method    type GetType()
RoundUpToUnit Method    System.UInt64 RoundUpToUnit(Microsoft.Exchange....
ToBytes    Method    System.UInt64 ToBytes()
ToGB       Method    System.UInt64 ToGB()
ToKB       Method    System.UInt64 ToKB()
ToMB       Method    System.UInt64 ToMB()
ToString   Method    string ToString(), string ToString(string forma...
ToTB       Method    System.UInt64 ToTB()
```

From the output shown in the previous screenshot, we can see this object supports several methods that can be used convert the value. To obtain the `MaxSendSize` value in megabytes, we can call the `ToMB` method:

```
[PS] C:\>$mailbox.MaxSendSize.Value.ToMB()
10
```

In a traditional shell, you would have to perform complex string parsing to extract this type of information, but PowerShell and the .NET Framework make this much easier. As you'll see over time, this is one of the reasons why PowerShell's object-based nature really outshines a typical text-based command shell.

An important thing to point about this last example is that it would not work if the mailbox had not had a custom `MaxSendSize` limitation configured. Nevertheless, this provides a good illustration of the process you'll want to use when you're trying to learn about an object's properties or methods.

Variable expansion in strings

As mentioned in the *Understanding command syntax and parameters* recipe in this chapter, PowerShell uses quoting rules to determine how variables should be handled inside a quoted string. When enclosing a simple variable inside a double-quoted string, PowerShell will expand that variable and replace the variable with the value of the string. Let's take a look at how this works by starting off with a simple example:

```
[PS] C:\>$name = "Bob"
[PS] C:\> "The user name is $name"
The user name is Bob
```

This is pretty straightforward. We stored the string value of "Bob" inside the `$name` variable. We then include the `$name` variable inside a double-quoted string that contains a message. When we hit return, the `$name` variable is expanded and we get back the message we expect to see on the screen.

Now let's try this with a more complex object. Let's say that we want to store an instance of a mailbox object in a variable and access the `PrimarySmtpAddress` property inside the quoted string:

```
[PS] C:\>$mailbox = Get-Mailbox testuser
[PS] C:\>"The email address is $mailbox.PrimarySmtpAddress"
The email address is test user.PrimarySmtpAddress
```

Notice here that when we try to access the `PrimarySmtpAddress` property of our mailbox object inside the double-quoted string, we're not getting back the information that we'd expect. This is a very common stumbling block when it comes to working with objects and properties inside strings. We can get around this using sub-expression notation. This requires that you enclose the entire object within `$ ()` characters inside the string:

```
[PS] C:\>"The email address is $($mailbox.PrimarySmtpAddress)"
The email address is testuser@contoso.com
```

Using this syntax, the `PrimarySmtpAddress` property of the `$mailbox` object is properly expanded and the correct information is returned. This technique will be useful later when extracting data from objects and generating reports or logfiles.

Strongly typed variables

PowerShell will automatically try to select the correct data type for a variable based on the value being assigned to it. You don't have to worry about doing this yourself, but we do have the ability to explicitly assign a type to a variable if needed. This is done by specifying the data type in square brackets before the variable name:

```
[string]$a = 32
```

Here we've assigned the value of 32 to the `$a` variable. Had we not strongly typed the variable using the `[string]` type shortcut, `$a` would have been created using the `Int32` data type, since the value we assigned was a number that was not enclosed in single or double quotes. Take a look at the following screenshot:

```

Machine: MBX1.CONTOSO.COM
[PS] C:\>$var1 = 32
[PS] C:\>$var1.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     Int32                                     System.ValueType

[PS] C:\>[string]$var2 = 32
[PS] C:\>$var2.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     String                                    System.Object

```

As you can see here, the `$var1` variable is initially created without any explicit typing. We use the `GetType()` method, which can be used on any object in the shell, to determine the data type of `$var1`. Since the value assigned was a number not enclosed in quotes, it was created using the `Int32` data type. When using the `[string]` type shortcut to create `$var2` with the same value, you can see that it has now been created as a string.

It is good to have an understanding of data types because when building scripts that return objects, you may need to have some control over this. For example, you may want to report on the amount of free disk space on an Exchange server. If we store this value in the property of a custom object as a string, we lose the ability to sort on that value. There are several examples throughout the book that use this technique.

See *Appendix A, Common Shell Information*, for a listing of commonly-used type shortcuts.

Formatting output

One of the most common PowerShell questions is how to get information returned from commands in the desired output on the screen. In this recipe, we'll take a look at how you can output data from commands and format that information for viewing on the screen.

How to do it...

To change the default output and view the properties of an object in list format, pipe the command to the `Format-List` cmdlet:

```
Get-Mailbox testuser | Format-List
```


To view specific properties in table format, supply a comma-separated list of property names as parameters, as shown next when using `Format-Table`:

```
Get-Mailbox testuser | Format-Table name,alias
```

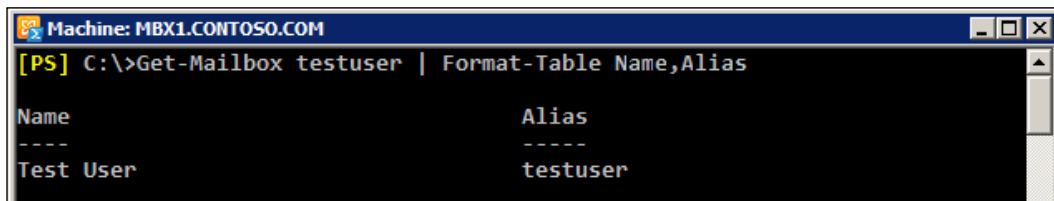
How it works...

When you run the `Get-Mailbox` cmdlet, you only see the `Name`, `Alias`, `ServerName`, and `ProhibitSendQuota` properties of each mailbox in a table format. This is because the `Get-Mailbox` cmdlet receives its formatting instructions from the `exchange.format.ps1xml` file located in the Exchange server bin directory.

PowerShell cmdlets use a variety of formatting files that usually include a default view with only a small subset of predefined properties. When you need to override the default view, you can use `Format-List` and `Format-Table` cmdlets.

You can also select specific properties with `Format-List`, just as we saw when using the `Format-Table` cmdlet. The difference is, of course, that the output will be displayed in list format.

Let's take a look at the output from the `Format-Table` cmdlet, as shown previously:

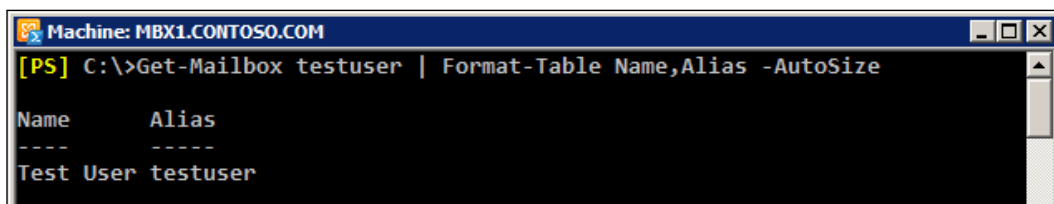


```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Mailbox testuser | Format-Table Name, Alias

Name                Alias
----                -
Test User           testuser
```

As you can see here, we get both properties of the mailbox formatted as a table.

When using `Format-Table` cmdlet, you may find it useful to use the `-AutoSize` parameter to organize the columns based on the width of the data:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>Get-Mailbox testuser | Format-Table Name, Alias -AutoSize

Name      Alias
----      -
Test User testuser
```

This command selects the same properties as our previous example, but this time we are using the `-AutoSize` parameter and the columns are adjusted to use only as much space on the screen as is needed. Remember, you can use the `ft` alias instead of typing the entire `Format-Table` cmdlet name. You can also use the `fl` alias for the `Format-List` cmdlet. Both of these aliases can keep your commands concise and are very convenient when working interactively in the shell.

There's more...

One thing to keep in mind is that you never want to use the `Format-*` cmdlets in the middle of a pipeline since most other cmdlets will not understand what to do with the output. The `Format-*` cmdlets should normally be the last thing you do in a command unless you are sending the output to a printer or a text file.

To send formatted output to a text file, you can use the `Out-File` cmdlet. In the following command, the `Format-List` cmdlet uses the asterisk (*) character as a wildcard and exports all of the property values for the mailbox to a text file:

```
Get-Mailbox testuser | fl * | Out-File c:\mb.txt
```

To add data to the end of an existing file, use the `-Append` parameter with the `Out-File` cmdlet. Even though we're using the `Out-File` cmdlet here, the traditional `cmd` output redirection operators such as `>` and `>>` can still be used. The difference is that the cmdlet gives you a little more control over the output method and provides parameters for tasks, including setting the encoding of the file.

You can sort the output of a command using the `Sort-Object` cmdlet. For example, this command will display all mailbox databases in alphabetical order:

```
Get-MailboxDatabase | sort name | ft name
```

We are using the `sort` alias for the `Sort-Object` cmdlet specifying `name` as the property we want to sort. To reverse the sort order, use the descending switch parameter:

```
Get-MailboxDatabase | sort name -desc | ft name
```

See also

- ▶ *The Understanding the pipeline recipe*
- ▶ *The Exporting reports to text and CSV files recipe in Chapter 2, Exchange Management Shell Common Tasks*

Working with arrays and hash tables

Like many other scripting and programming languages, Windows PowerShell allows you to work with arrays and hash tables. An array is a collection of values that can be stored in a single object. A hash table is also known as an associative array, and is a dictionary that stores a set of key-value pairs. You'll need to have a good grasp of arrays so that you can effectively manage objects in bulk and gain maximum efficiency in the shell. In this recipe, we'll take a look at how we can use both types of arrays to store and retrieve data.

How to do it...

You can initialize an array that stores a set of items by assigning multiple values to a variable. All you need to do is separate each value with a comma. The following command would create an array of server names:

```
$servers = "EX1", "EX2", "EX3"
```

To create an empty hash table, use the following syntax:

```
$hashtable = @{ }
```

Now that we have an empty hash table, we can add key-value pairs:

```
$hashtable["server1"] = 1
```

```
$hashtable["server2"] = 2
```

```
$hashtable["server3"] = 3
```

Notice in this example that we can assign a value based on a key name, not using an index number as we saw with a regular array. Alternatively, we can create this same object using a single command using the following syntax:

```
$hashtable = @{server1 = 1; server2 = 2; server3 = 3}
```

You can see here that we used a semicolon (;) to separate each key-value pair. This is only required if the entire hash table is created in one line.

You can break this up into multiple lines to make it easier to read:

```
$hashtable = @{  
    server1 = 1  
    server2 = 2  
    server3 = 3  
}
```

How it works...

Let's start off by looking at how arrays work in PowerShell. When working with arrays, you can access specific items and add or remove elements. In our first example, we assigned a list of server names to the `$servers` array. To view all of the items in the array, simply type the variable name and hit return:

```
[PS] C:\>$servers
```

EX1

EX2

EX3

Array indexing allows you to access a specific element of an array using its index number inside square brackets (`[]`). PowerShell arrays are zero-based, meaning that the first item in the array starts at index zero. For example, use the second index to access the third element of the array, as shown next:

```
[PS] C:\>$servers[2]
```

EX3

To assign a value to a specific element of the array, use the equals (=) assignment operator. We can change the value from the last example using the following syntax:

```
[PS] C:\>$servers[2] = "EX4"
```

```
[PS] C:\>$servers[2]
```

EX4

Let's add another server to this array. To append a value, use the plus equals (+=) assignment operator as shown here:

```
[PS] C:\>$servers += "EX5"
```

```
[PS] C:\>$servers
```

EX1

EX2

EX4

EX5

To determine how many items are in an array, we can access the `Count` property to retrieve the total number of array elements:

```
[PS] C:\>$servers.Count
```

4

We can loop through each element in the array with the `ForEach-Object` cmdlet and display the value in a string:

```
$servers | ForEach-Object {"Server Name: $_"}
```

We can also check for a value in an array using the `-Contains` or `-NotContains` conditional operators:

```
[PS] C:\>$servers -contains "EX1"
True
```

In this example, we are working with a one-dimensional array, which is what you'll commonly be dealing with in the Exchange Management Shell. PowerShell supports more complex array types such as jagged and multidimensional arrays, but these are beyond the scope of what you'll need to know for the examples in this book.

Now that we've figured out how arrays work, let's take a closer look at hash tables. When viewing the output for a hash table, the items are returned in no particular order. You'll notice this when viewing the hash table we created earlier:

```
[PS] C:\>$hashtable

Name                Value
----                -
server2             2
server1             1
server3             3
```

If you want to sort the hash table, you can call the `GetEnumerator` method and sort by the `Value` property:

```
[PS] C:\>$hashtable.GetEnumerator() | sort value

Name                Value
----                -
server1             1
server2             2
server3             3
```

Hash tables can be used when creating custom objects, or to provide a set of parameter names and values using parameter splatting. Instead of specifying parameter names one by one with a cmdlet, you can use a hash table with keys that match the parameter's names and their associated values will automatically be used for input:

```
$parameters = @{
    Title = "Manager"
    Department = "Sales"
```

```

    Office = "Headquarters"
}
Set-User testuser @parameters

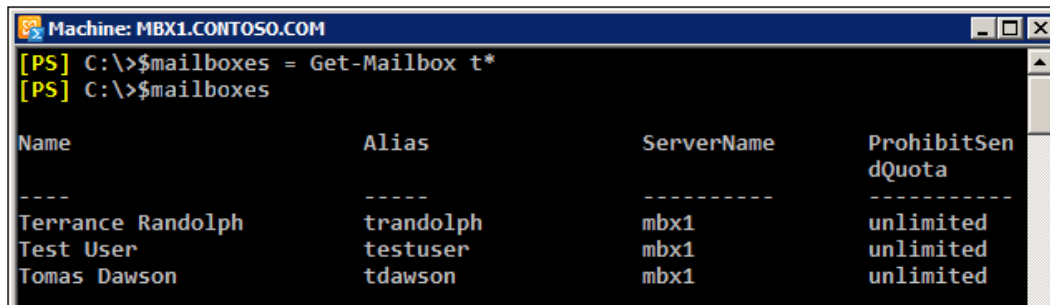
```

This command automatically populates the parameter values for Title, Department, and Office when running the Set-User cmdlet for the testuser mailbox.

For more details and examples for working with hash tables, run Get-Help about_Hash_Tables.

There's more...

You can think of a collection as an array created from the output of a command. For example, the Get-Mailbox cmdlet can be used to create an object that stores a collection of mailboxes, and we can work with this object just as we would with any other array. You'll notice that, when working with collections, such as a set of mailboxes, you can access each mailbox instance as an array element. Consider the following example:



```

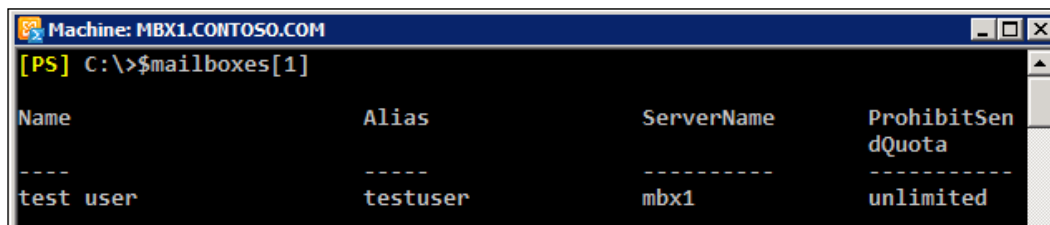
Machine: MBX1.CONTOSO.COM
[PS] C:\>$mailboxes = Get-Mailbox t*
[PS] C:\>$mailboxes

```

Name	Alias	ServerName	ProhibitSendQuota
Terrance Randolph	trandolph	mbx1	unlimited
Test User	testuser	mbx1	unlimited
Tomas Dawson	tdawson	mbx1	unlimited

First, we retrieve a list of mailboxes that start with the letter `t` and assign that to the `$mailboxes` variable. From looking at the items in the `$mailboxes` object, we can see that the `testuser` mailbox is the second mailbox in the collection.

Since arrays are zero-based, we can access that item using the first index, as shown next:



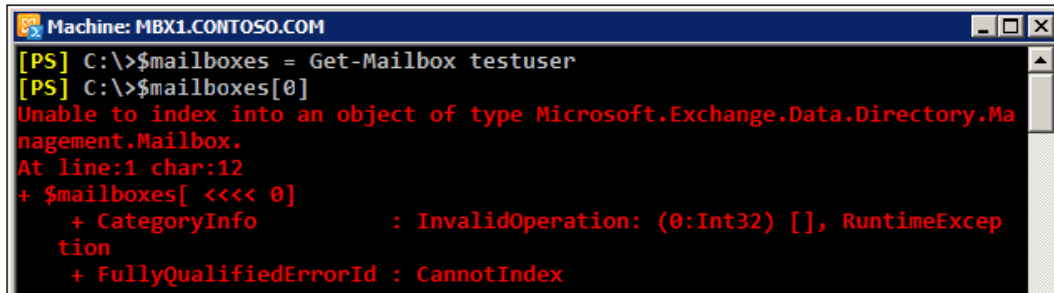
```

Machine: MBX1.CONTOSO.COM
[PS] C:\>$mailboxes[1]

```

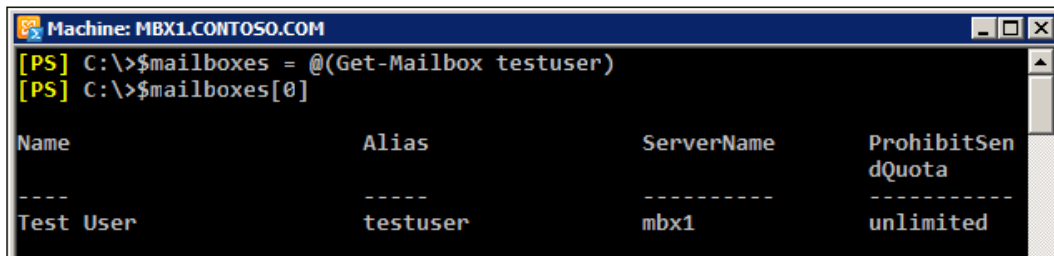
Name	Alias	ServerName	ProhibitSendQuota
test user	testuser	mbx1	unlimited

If your command only returns one item, then the output can no longer be accessed using array notation. In the following example, the `$mailboxes` object contains only one mailbox and will display an error when trying to access an item using array notation:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>$mailboxes = Get-Mailbox testuser
[PS] C:\>$mailboxes[0]
Unable to index into an object of type Microsoft.Exchange.Data.Directory.Management.Mailbox.
At line:1 char:12
+ $mailboxes[ <<<< 0]
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (0:Int32) [], RuntimeException
+ FullyQualifiedErrorId : CannotIndex
```

Even though it will only store one item, you can initialize this object as an array, using the following syntax:



```
Machine: MBX1.CONTOSO.COM
[PS] C:\>$mailboxes = @(Get-Mailbox testuser)
[PS] C:\>$mailboxes[0]

Name                Alias                ServerName            ProhibitSendQuota
----                -
Test User           testuser             mbx1                  unlimited
```

You can see here that we've wrapped the command inside the `@()` characters to ensure that PowerShell will always interpret the `$mailboxes` object as an array. This can be useful when you're building a script that always needs to work with an object as an array, regardless of the number of items returned from the command that created the object. Since the `$mailboxes` object has been initialized as an array, you can add and remove elements as needed.

We can also add and remove items to multi-valued properties, just as we would with a normal array. To add an e-mail address to the `testuser` mailbox, we can use the following commands:

```
$mailbox = Get-Mailbox testuser
$mailbox.EmailAddresses += "testuser@contoso.com"
Set-Mailbox testuser -EmailAddresses $mailbox.EmailAddresses
```

In this example, we created an instance of the `testuser` mailbox by assigning the command to the `$mailbox` object. We can then work with the `EmailAddresses` property to view, add, and remove e-mail addresses from this mailbox. You can see here that the plus equals (`+=`) operator was used to append a value to the `EmailAddresses` property.

We can also remove that value using the minus equals (-=) operator:

```
$mailbox.EmailAddresses -= "testuser@contoso.com"
Set-Mailbox testuser -EmailAddresses $mailbox.EmailAddresses
```



There is actually an easier way to add and remove e-mail addresses on recipient objects. See *Adding and removing recipient e-mail addresses* in *Chapter 3, Managing Recipients* for details.

We've covered the core concepts in this section that you'll need to know when working with arrays. For more details run `Get-Helpabout_arrays`.

See also

- ▶ The *Working with variables and objects* recipe
- ▶ The *Creating custom objects* recipe

Looping through items

Loop processing is a concept that you will need to master in order to write scripts and one-liners with efficiency. You'll need to use loops to iterate over each item in an array or a collection of items, and then run one or more commands within a script block against each of those objects. In this recipe, we'll take a look at how you can use `foreach` loops and the `ForEach-Object` cmdlet to process items in a collection.

How to do it...

The `foreach` statement is a language construct used to iterate through values in a collection of items. The following example shows the syntax used to loop through a collection of mailboxes, returning only the name of each mailbox:

```
foreach($mailbox in Get-Mailbox) {$mailbox.Name}
```

In addition, you can take advantage of the PowerShell pipeline and perform loop processing using the `ForEach-Object` cmdlet. This example produces the same result as the one shown previously:

```
Get-Mailbox | ForEach-Object {$_.Name}
```

You will often see the given command written using an alias of the `ForEach-Object` cmdlet, such as the percent sign (%):

```
Get-Mailbox | %{$_.Name}
```


How it works...

The first part of a `foreach` statement is enclosed in parenthesis and represents a variable and a collection. In the previous example, the collection is the list of mailboxes returned from the `Get-Mailbox` cmdlet. The script block contains the commands that will be run for every item in the collection of mailboxes. Inside the script block, the `$mailbox` object is assigned the value of the current item being processed in the loop. This allows you to access each mailbox one at a time using the `$mailbox` variable.

When you need to perform loop processing within a pipeline, you can use the `ForEach-Object` cmdlet. The concept is similar, but the syntax is different because objects in the collection are coming across the pipeline.

The `ForEach-Object` cmdlet allows you to process each item in a collection using the `$_` automatic variable, which represents the current object in the pipeline. The `ForEach-Object` cmdlet is probably one of the most commonly-used cmdlets in PowerShell, and we'll rely on it heavily in many examples throughout the book.

The code inside the script block used with both looping methods can be more complex than just a simple expression. The script block can contain a series of commands or an entire script. Consider the following code:

```
Get-MailboxDatabase -Status | %{  
    $DBName = $_.Name  
    $whiteSpace = $_.AvailableNewMailboxSpace.ToMb()  
    "The $DBName database has $whiteSpace MB of total white space"  
}
```

In this example, we're looping through each mailbox database in the organization using the `ForEach-Object` cmdlet. Inside the script block, we've created multiple variables, calculated the total megabytes of whitespace in each database, and returned a custom message that includes the database name and corresponding whitespace value. This is a simple example, but keep in mind that inside the script block you can run other cmdlets, work with variables, create custom objects, and more.

PowerShell also supports other language constructs for processing items such as `for`, `while`, and `do` loops. Although these can be useful in some cases, we won't rely on them much for the remaining examples in this book. You can read more about them and view examples using the `get-help about_for`, `get-help about_while`, and `get-help about_do` commands in the shell.

There's more...

There are some key differences about the `foreach` statement and the `ForEach-Object` cmdlet that you'll want to be aware of when you need to work with loops. First, the `ForEach-Object` cmdlet can process one object at a time as it comes across the pipeline. When you process a collection using the `foreach` statement, this is the exact opposite. The `foreach` statement requires that all of the objects that need to be processed within a loop are collected and stored in memory before processing begins. We'll want to take advantage of the PowerShell pipeline and its streaming behavior whenever possible since it is much more efficient.

The other thing to take note of is that in PowerShell, `foreach` is not only a keyword, but also an alias. This can be a little counterintuitive, especially when you are new to PowerShell and you run into a code sample that uses the following syntax:

```
Get-Mailbox | foreach {$_ .Name}
```

At first glance, this might seem like we're using the `foreach` keyword, but we're actually using an alias for the `ForEach-Object` cmdlet. The easiest way to remember this distinction is that the `foreach` language construct is always used before a pipeline. If you use `foreach` after a pipeline, PowerShell will use the `foreach` alias which corresponds to the `ForEach-Object` cmdlet.

See also

- ▶ The *Working with arrays and hash tables* recipe
- ▶ The *Understanding the pipeline* recipe
- ▶ The *Creating custom objects* recipe

Creating and running scripts

You can accomplish many tasks by executing individual cmdlets or running multiple commands in a pipeline, but there may be times where you want to create a script that performs a series of operations or that loads a library of functions and predefined variables and aliases into the shell. In this recipe, we'll take a look at how you can create and run scripts in the shell.

How to do it...

1. Let's start off by creating a basic script that automates a multi-step process. We'll start up a text editor, such as Notepad, and enter the following code:

```
param(
    $name,
    $maxsendsize,
    $maxreceivesize,
    $city,
    $state,
    $title,
    $department
)

Set-Mailbox -Identity $name `
-MaxSendSize $maxsendsize `
-MaxReceiveSize $maxreceivesize

Set-User -Identity $name `
-City $city `
-StateOrProvince $state `
-Title $title `
-Department $department

Add-DistributionGroupMember -Identity DL_Sales `
-Member $name
```

2. Next, we'll save the file on the C:\ drive using the name Update-SalesMailbox.ps1.
3. We can then run this script and provide input using parameters that have been declared using the param keyword:

```
C:\Update-SalesMailbox.ps1 -name testuser `
-maxsendsize 25mb `
-maxreceivesize 25mb `
-city Phoenix `
-state AZ `
-title Manager `
-department Sales
```