After completing this chapter, you will be able to

- Use Windows PowerShell remoting to connect to a remote system.
- Use Windows PowerShell remoting to run commands on a remote system.
- Use Windows PowerShell jobs to run commands in the background.
- Receive the results of background jobs.
- Keep the results of background jobs.

Understanding Windows PowerShell remoting

The configuration of Windows PowerShell remoting on the server side is easy—it just works. On the client side, it must first be enabled, and then it also just works. When talking about Windows PowerShell remoting, a bit of confusion can arise because there are several different ways of running commands on remote servers. Depending on your particular network configuration and security needs, one or more methods of remoting might not be appropriate.

Classic remoting

Classic remoting in Windows PowerShell relies on protocols such as DCOM and RPC to make connections to remote machines. Traditionally, these protocols require opening many ports in the firewall and starting various services that the different cmdlets use. To find the Windows PowerShell cmdlets that natively support remoting, use the Get-Help cmdlet. Specify a value of computername for the -Parameter parameter of the Get-Help cmdlet. This command produces a nice list of all cmdlets that have native support for remoting. The command and associated output are shown on the following page.
<table>
<thead>
<tr>
<th>Name</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add-Computer</td>
<td>Add the local computer to a domain or workgroup.</td>
</tr>
<tr>
<td>Clear-EventLog</td>
<td>Deletes all entries from specified event logs on the local or remote computers.</td>
</tr>
<tr>
<td>Connect-PSSession</td>
<td>Reconnects to disconnected sessions.</td>
</tr>
<tr>
<td>Connect-WSMan</td>
<td>Connects to the WinRM service on a remote computer.</td>
</tr>
<tr>
<td>Disconnect-WSMan</td>
<td>Disconnects the client from the WinRM service on a remote computer.</td>
</tr>
<tr>
<td>Enter-PSSession</td>
<td>Starts an interactive session with a remote computer.</td>
</tr>
<tr>
<td>Get-Counter</td>
<td>Gets performance counter data from local and remote computers.</td>
</tr>
<tr>
<td>Get-EventLog</td>
<td>Gets the events in an event log, or a list of the event logs, on the local or remote computers.</td>
</tr>
<tr>
<td>Get-HotFix</td>
<td>Gets the hotfixes that have been applied to the local and remote computers.</td>
</tr>
<tr>
<td>Get-Process</td>
<td>Gets the processes that are running on the local computer or a remote computer.</td>
</tr>
<tr>
<td>Get-PSSession</td>
<td>Gets the Windows PowerShell sessions on local and remote computers.</td>
</tr>
<tr>
<td>Get-Service</td>
<td>Gets the services on a local or remote computer.</td>
</tr>
<tr>
<td>Get-WinEvent</td>
<td>Gets events from event logs and event tracing log files on local and remote computers.</td>
</tr>
<tr>
<td>Get-WmiObject</td>
<td>Gets instances of Windows Management Instrumentation (WMI) classes or information about the available classes.</td>
</tr>
<tr>
<td>Get-WSManInstance</td>
<td>Displays management information for a resource instance specified by a Resource URI.</td>
</tr>
<tr>
<td>Invoke-Command</td>
<td>Runs commands on local and remote computers.</td>
</tr>
<tr>
<td>Invoke-WmiMethod</td>
<td>Calls Windows Management Instrumentation (WMI) methods.</td>
</tr>
<tr>
<td>Invoke-WSManAction</td>
<td>Invokes an action on the object that is specified by the Resource URI and by the selectors.</td>
</tr>
<tr>
<td>Limit-EventLog</td>
<td>Sets the event log properties that limit the size of the event log and the age of its entries.</td>
</tr>
<tr>
<td>New-EventLog</td>
<td>Creates a new event log and a new event source on a local or remote computer.</td>
</tr>
<tr>
<td>New-PSSession</td>
<td>Creates a persistent connection to a local or remote computer.</td>
</tr>
<tr>
<td>New-WSManInstance</td>
<td>Creates a new instance of a management resource.</td>
</tr>
<tr>
<td>Receive-Job</td>
<td>Gets the results of the Windows PowerShell background jobs in the current session.</td>
</tr>
<tr>
<td>Receive-PSSession</td>
<td>Gets results of commands in disconnected sessions</td>
</tr>
<tr>
<td>Register-WmiEvent</td>
<td>Subscribes to a Windows Management Instrumentation (WMI) event.</td>
</tr>
<tr>
<td>Remove-Computer</td>
<td>Removes the local computer from its domain.</td>
</tr>
<tr>
<td>Remove-EventLog</td>
<td>Deletes an event log or unregisters an event source.</td>
</tr>
<tr>
<td>Remove-PSSession</td>
<td>Closes one or more Windows PowerShell sessions (PSSessions).</td>
</tr>
<tr>
<td>Remove-WmiObject</td>
<td>Deletes an instance of an existing Windows Management Instrumentation (WMI) class.</td>
</tr>
<tr>
<td>Remove-WSManInstance</td>
<td>Deletes a management resource instance.</td>
</tr>
<tr>
<td>Rename-Computer</td>
<td>Renames a computer.</td>
</tr>
<tr>
<td>Restart-Computer</td>
<td>Restarts (&quot;reboots&quot;) the operating system on local and remote computers.</td>
</tr>
<tr>
<td>Set-Service</td>
<td>Starts, stops, and suspends a service, and changes its properties.</td>
</tr>
<tr>
<td>Set-WmiInstance</td>
<td>Creates or updates an instance of an existing Windows Management Instrumentation (WMI) class.</td>
</tr>
</tbody>
</table>
Set-WSManInstance     Modifies the management information that is related to a resource.
Show-EventLog         Displays the event logs of the local or a remote computer in Event Viewer.
Stop-Computer         Stops (shuts down) local and remote computers.
Test-Connection       Sends ICMP echo request packets ("pings") to one or more computers.
Test-WSMan            Tests whether the WinRM service is running on a local or remote computer.
Write-EventLog        Writes an event to an event log.

As you can tell, many of the Windows PowerShell cmdlets that have the -ComputerName parameter relate to Web Services Management (WSMAN), Common Information Model (CIM), or sessions. To remove these cmdlets from the list, modify the command a bit to use Where-Object (?) is an alias for Where-Object). The revised command and associated output are shown here.

PS C:\> Get-Help * -Parameter computername -Category Cmdlet | ? modulename -match 'PowerShell.Management' | Sort-Object name | Format-Table name, synopsis -AutoSize -Wrap

<table>
<thead>
<tr>
<th>Name</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add-Computer</td>
<td>Add the local computer to a domain or workgroup.</td>
</tr>
<tr>
<td>Clear-EventLog</td>
<td>Deletes all entries from specified event logs on the local or remote computers.</td>
</tr>
<tr>
<td>Get-EventLog</td>
<td>Gets the events in an event log, or a list of the event logs, on the local or remote computers.</td>
</tr>
<tr>
<td>Get-HotFix</td>
<td>Gets the hotfixes that have been applied to the local and remote computers.</td>
</tr>
<tr>
<td>Get-Process</td>
<td>Gets the processes that are running on the local computer or a remote computer.</td>
</tr>
<tr>
<td>Get-Service</td>
<td>Gets the services on a local or remote computer.</td>
</tr>
<tr>
<td>Get-WmiObject</td>
<td>Gets instances of Windows Management Instrumentation (WMI) classes or information about the available classes.</td>
</tr>
<tr>
<td>Invoke-WmiMethod</td>
<td>Calls Windows Management Instrumentation (WMI) methods.</td>
</tr>
<tr>
<td>Limit-EventLog</td>
<td>Sets the event log properties that limit the size of the event log and the age of its entries.</td>
</tr>
<tr>
<td>New-EventLog</td>
<td>Creates a new event log and a new event source on a local or remote computer.</td>
</tr>
<tr>
<td>Register-WmiEvent</td>
<td>Subscribes to a Windows Management Instrumentation (WMI) event.</td>
</tr>
<tr>
<td>Remove-Computer</td>
<td>Removes the local computer from its domain.</td>
</tr>
<tr>
<td>Remove-EventLog</td>
<td>Deletes an event log or unregisters an event source.</td>
</tr>
<tr>
<td>Remove-WmiObject</td>
<td>Deletes an instance of an existing Windows Management Instrumentation (WMI) class.</td>
</tr>
<tr>
<td>Rename-Computer</td>
<td>Renames a computer.</td>
</tr>
<tr>
<td>Restart-Computer</td>
<td>Restarts (&quot;reboots&quot;) the operating system on local and remote computers.</td>
</tr>
<tr>
<td>Set-Service</td>
<td>Starts, stops, and suspends a service, and changes its properties.</td>
</tr>
<tr>
<td>Set-WmiInstance</td>
<td>Creates or updates an instance of an existing Windows Management Instrumentation (WMI) class.</td>
</tr>
<tr>
<td>Show-EventLog</td>
<td>Displays the event logs of the local or a remote computer in Event Viewer.</td>
</tr>
<tr>
<td>Stop-Computer</td>
<td>Stops (shuts down) local and remote computers.</td>
</tr>
<tr>
<td>Test-Connection</td>
<td>Sends ICMP echo request packets (&quot;pings&quot;) to one or more computers.</td>
</tr>
<tr>
<td>Write-EventLog</td>
<td>Writes an event to an event log.</td>
</tr>
</tbody>
</table>
Some of the cmdlets provide the ability to specify credentials. This allows you to use a different user account to make the connection and to retrieve the data. Figure 4-1 displays the credential dialog box that appears when such a cmdlet runs.

**FIGURE 4-1** Cmdlets that support the `-Credential` parameter prompt for credentials when supplied with a user name.

This technique of using the `-ComputerName` and `-Credential` parameters in a cmdlet is shown here.

PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName ex1 -Credential nwtraders\administrator

```
TimeCreated                  ProviderName              Id Message
-----------                  ------------         -- -------
7/1/2015 11:54:14 AM        MSExchange ADAccess 2080 Process MAD.EXE (...)
```

However, as mentioned earlier, use of these cmdlets often requires opening holes in the firewall or starting specific services. By default, these types of cmdlets fail when they are run on remote machines that don’t have relaxed access rules. An example of this type of error is shown here.

PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential nwtraders\administrator

```
Get-WinEvent : The RPC server is unavailable
At line:1 char:1
+ Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Cre ...
```

```
+ CategoryInfo : NotSpecified: (:) [Get-WinEvent], EventLogException
```
Other cmdlets, such as `Get-Service` and `Get-Process`, do not have a `-Credential` parameter, and therefore the commands associated with cmdlets such as `Get-Service` or `Get-Process` impersonate the logged-on user. Such a command is shown here.

```
PS C:\> Get-Service -ComputerName hyperv -Name bits
```

<table>
<thead>
<tr>
<th>Status</th>
<th>Name</th>
<th>DisplayName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>bits</td>
<td>Background Intelligent Transfer Ser...</td>
</tr>
</tbody>
</table>

The fact that a cmdlet does not support alternate credentials does not mean that the cmdlet must impersonate the logged-on user. Holding down the Shift key and right-clicking the Windows PowerShell icon on the taskbar brings up a menu from which you can select commands to run the program as a different user. This menu is shown in Figure 4-2.

![Figure 4-2](image)

**FIGURE 4-2** You can use commands on the menu from the Windows PowerShell console to run with different security credentials.

The Run As Different User dialog box is shown in Figure 4-3.
You can use the Run As Different User dialog box to enter a different user context.

Using the Run As Different User dialog box makes alternative credentials available for Windows PowerShell cmdlets that do not support the `-Credential` parameter.

**WinRM**
Beginning with Windows Server 2012, Windows Server installs with Windows Remote Management (WinRM) configured and running to support remote Windows PowerShell commands. WinRM is the Microsoft implementation of the industry-standard WS-Management protocol. As such, WinRM provides a firewall-friendly method of accessing remote systems in an interoperable manner. It is the remoting mechanism used by the CIM cmdlets. As soon as your Windows Server computer is up and running, you can make a remote connection and run commands, or open an interactive Windows PowerShell console. In Windows 10, on the other hand, WinRM is locked down. Therefore, the first step is to use the `Enable-PSRemoting` cmdlet to configure Windows PowerShell remoting on the client machine. When `Enable-PSRemoting` is run, it performs the following steps:

1. Starts the WinRM service
2. Sets the WinRM service startup type to Automatic
3. Creates a listener to accept requests from any IP address
4. Enables inbound firewall exceptions for WSMAN traffic

5. Sets a target listener named Microsoft.powershell

6. Sets a target listener named Microsoft.powershell.workflow

7. Sets a target listener named Microsoft.powershell32 on 64-bit computers

8. Enables all session configurations

9. Changes the security descriptor of all session configurations to allow remote access

10. Restarts the WinRM service to make the changes effective

When Enable-PSRemoting is run, the cmdlet prompts you to agree to performing the specified action. If you are familiar with the steps the cmdlet performs and you do not make any changes from the defaults, you can run the command by using the -Force switch parameter, and it will not prompt prior to making the changes. The syntax of this command is shown here.

Enable-PSRemoting -force

The use of the Enable-PSRemoting function in interactive mode is shown here, along with all associated output from the command.

PS C:\> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer by using the Windows Remote Management (WinRM) service.
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service startup type to Automatic
  3. Creating a listener to accept requests on any IP address
  4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic (for http only).

Do you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
WinRM is already set up to receive requests on this computer.
WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.
WinRM firewall exception enabled.

Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell SDDL:
O:NSG:BAD:P(A;=GA;=`BA)(A;=GA;=`IU)(A;=GA;=`RM)S:P(AU;FA;GA;=`WD)(AU;SA;GXGw;=`WD).
This lets selected users remotely run Windows PowerShell commands on this computer.".

[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
When Windows PowerShell remoting has been configured, use the `Test-WSMan` cmdlet to ensure that the WinRM remoting is properly configured and is accepting requests. A properly configured system replies with the information shown here.

```
PS C:\> Test-WSMan -ComputerName c10

wsnid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

This cmdlet also works with Windows PowerShell 5.0 remoting. The output shown here is from a domain controller running Windows Server 2012 R2 with Windows PowerShell 5.0 installed and WinRM configured for remote access.

```
PS C:\> Test-WSMan -ComputerName DC1

wsnid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```
If WinRM is not configured, an error returns from the system. Such an error from a Windows 8 client is shown here.

```
PS C:\> Test-WSMan -ComputerName w8c10
Test-WSMan : <f:WSManFault
Machine="w8c504.nwtraders.net"><f:Message>WinRM cannot complete the operation. Verify
that the specified computer name is valid, that the computer is accessible over the
network, and that a firewall exception for the WinRM service is enabled and allows
access from this computer. By default, the WinRM firewall exception for public
profiles limits access to remote computers within the same local subnet.
</f:Message></f:WSManFault>
At line:1 char:1
+ Test-WSMan -ComputerName w8c10
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (w8c10:String) [Test-WSMan], Invalid
      OperationException
    + FullyQualifiedErrorId : WsManError,Microsoft.WSMan.Management.TestWSManCommand
```

Keep in mind that configuring WinRM via the `Enable-PSRemoting` cmdlet does not enable the `Remote Management` firewall exception, and therefore PING commands will not work by default when pinging to a Windows 8 client system. This is shown here.

```
PS C:\> ping w8c504
Pinging w8c504.nwtraders.net [192.168.0.56] with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.56:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss).
```

Pings to a Windows Server 2012 R2 server, however, do work. This is shown here.

```
PS C:\> ping dc1
Pinging dc1.nwtraders.com [192.168.10.1] with 32 bytes of data:
Reply from 192.168.10.1: bytes=32 time<1ms TTL=128
Reply from 192.168.10.1: bytes=32 time<1ms TTL=128
Reply from 192.168.10.1: bytes=32 time<1ms TTL=128
Reply from 192.168.10.1: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.10.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\>
```
Creating a remote Windows PowerShell session

For simple configuration on a single remote machine, entering a remote Windows PowerShell session is the answer. To enter a remote Windows PowerShell session, use the `Enter-PSSession` cmdlet. This creates an interactive remote Windows PowerShell session on a target machine and uses the default remote endpoint. If you do not supply credentials, the remote session impersonates the currently logged-on user. The output shown here illustrates connecting to a remote computer named dc1. After the connection is established, the Windows PowerShell prompt changes to include the name of the remote system. `Set-Location` (which has an alias of `sl`) changes the working directory on the remote system to `C:\`. Next, the `Get-CimInstance` cmdlet retrieves the BIOS information for the remote system. The `exit` command exits the remote session, and the Windows PowerShell prompt returns to the prompt configured previously.

```
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> sl c:\
[dc1]: PS C:\> gcim win32_bios

SMBIOSBIOSVersion : Hyper-V UEFI Release v1.0
Manufacturer       : Microsoft Corporation
Name               : Hyper-V UEFI Release v1.0
SerialNumber       : 3601-6926-9922-0181-5225-8175-58
Version            : VIRTUAL - 1

[dc1]: PS C:\> exit
PS C:\>
```

The good thing is that when you use the Windows PowerShell transcript tool via `Start-Transcript`, the transcript tool captures output from the remote Windows PowerShell session, in addition to output from the local session. Indeed, all commands entered appear in the transcript. The following commands illustrate beginning a transcript, entering a remote Windows PowerShell session, and stopping the transcript.

```
PS C:\> Start-Transcript
Transcript started, output file is C:\Users\administrator\Documents\PowerShell_transcript.C10.DyQ6Wy5p.20150711150938.txt
PS C:\> Enter-PSSession -ComputerName dc1
PS C:\> Stop-Transcript
Transcript stopped, output file is C:\Users\administrator\Documents\PowerShell_transcript.C10.DyQ6Wy5p.20150711150938.txt
```

Figure 4-4 displays a copy of the transcript from the previous session.
FIGURE 4-4 The Windows PowerShell transcript tool records commands and output received from a remote Windows PowerShell session.

If you anticipate making multiple connections to a remote system, use the `New-PSSession` cmdlet to create a remote Windows PowerShell session. You can use `New-PSSession` to store the remote session in a variable and to enter and leave the remote session as often as required—without the additional overhead of creating and destroying remote sessions. In the commands that follow, a new Windows PowerShell session is created via the `New-PSSession` cmdlet. The newly created session is stored in the `$dc1` variable. Next, the `Enter-PSSession` cmdlet is used to enter the remote session by using the stored session. A command retrieves the remote hostname, and the remote session is exited via the `exit` command. Next, the session is re-entered, and the last process is retrieved. The session is exited again. Finally, the `Get-PSSession` cmdlet retrieves Windows PowerShell sessions on the system, and all sessions are removed via the `Remove-PSSession` cmdlet.

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential nwtraders\administrator
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> hostname
dc1
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> gps | select -Last 1
Handles  NPM(K)  PM(K)  WS(K)  VM(M)  CPU(s)  Id  ProcessName
-------  ------  -----  -----  -----  ------  --  -----------
292       9  39536  50412  158    1.97    2332 wsmprovhost
```
Running a single Windows PowerShell command

If you have a single command to run, it does not make sense to go through all the trouble of building and entering an interactive remote Windows PowerShell session. Instead of creating a remote Windows PowerShell console session, you can run a single command by using the `Invoke-Command` cmdlet. If you have a single command to run, use the cmdlet directly and specify the computer name and any credentials required for the connection. You are still creating a remote session, but you are also removing the session. Therefore, if you have a lot of commands to run against the remote machine, a performance problem could arise. But for single commands, this technique works well. The technique is shown here, where the last process running on the Ex1 remote server is shown.

```
PS C:\> Invoke-Command -ComputerName ex1 -ScriptBlock {gps | select -Last 1}
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName   PSComputerName
-------  ------    -----      ----- -----   ------     -- -----------   ------------
  224      34    47164      51080   532     0.58  10164 wsmprovhost   ex1
```

If you have several commands, or if you anticipate making multiple connections, the `Invoke-Command` cmdlet accepts a session name or a session object in the same manner as the `Enter-PSSession` cmdlet. In the output shown here, a new PSSession is created to a remote computer named dc1. The remote session is used to retrieve two different pieces of information. When the Windows PowerShell remote session is completed, the session stored in the `$dc1` variable is explicitly removed.

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential nwtraders\administrator
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {hostname}
dc1
PS C:\> Remove-PSSession $dc1
```
By using *Invoke-Command*, you can run the same command against a large number of remote systems. The secret behind this power is that the `-ComputerName` parameter from the *Invoke-Command* cmdlet accepts an array of computer names. In the output shown here, an array of computer names is stored in the variable `$cn`. Next, the `$cred` variable holds the *PSCredential* object for the remote connections. Finally, the *Invoke-Command* cmdlet is used to make connections to all of the remote machines and to return the BIOS information from the systems. The nice thing about this technique is that an additional property, *PSComputerName*, is added to the returning object, so you can easily identify which BIOS is associated with which computer system. The commands and associated output are shown here.

```
PS C:\> $cn = "dc1","dc3","ex1","sql1","wsus1","wds1","hyperv1","hyperv2","hyperv3"
PS C:\> $cred = get-credential nwtraders\administrator
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}
```

```
Manufacturer      : Intel Corp.
Name              : BIOS Date: 09/27/11 14:25:42 Ver: 04.06.04
SerialNumber      :
Version           : INTEL - 1072009
PSComputerName    : hyperv3

SMBIOSBIOSVersion : A11
Manufacturer      : Dell Inc.
Name              : Phoenix ROM BIOS PLUS Version 1.10 A11
SerialNumber      : BDY911.1
Version           : DELL - 15
PSComputerName    : hyperv2

SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQLS21
Version           : DELL - 6
PSComputerName    : dc1

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 3692-0963-1044-7503-9631-2546-83
Version           : VRTUAL - 3000919
PSComputerName    : wsus1

SMBIOSBIOSVersion : V1.6
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : To Be Filled By O.E.M.
Version           : 7583MS - 20091228
PSComputerName    : hyperv1
```
Using Windows PowerShell jobs

Windows PowerShell jobs can be used to run one or more commands in the background. After you start the Windows PowerShell job, the Windows PowerShell console returns immediately for further use, so you can accomplish multiple tasks at the same time. You can begin a new Windows PowerShell job by using the `Start-Job` cmdlet. The command you want to run as a job is placed in a script block, and the jobs are sequentially named Job1, Job2, and so on. This is shown here.

```powershell
PS C:\> Start-Job -ScriptBlock {get-process}
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
10     Job10           BackgroundJob   Running       True            localhost
```

PS C:\>
The jobs receive job IDs that are also sequentially numbered. The first job created in a Windows PowerShell console always has a job ID of 1. You can use either the job ID or the job name to obtain information about the job. This is shown here.

```powershell
PS C:\> Get-Job -Name job10
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Job10</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

```powershell
PS C:\> Get-Job -Id 10
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Job10</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

```powershell
PS C:\>
```

When you notice that the job has completed, you can receive the job. The `Receive-Job` cmdlet returns the same information that returns if a job is not used. The Job1 output is shown here (truncated to save space).

```powershell
PS C:\> Receive-Job -Name job10
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>9</td>
<td>1672</td>
<td>6032</td>
<td>80</td>
<td>0.00</td>
<td>1408</td>
<td>apdproxy</td>
</tr>
<tr>
<td>132</td>
<td>9</td>
<td>2316</td>
<td>5632</td>
<td>62</td>
<td></td>
<td>1364</td>
<td>atiec1xx</td>
</tr>
<tr>
<td>122</td>
<td>7</td>
<td>1716</td>
<td>4232</td>
<td>32</td>
<td></td>
<td>948</td>
<td>atiesrxx</td>
</tr>
<tr>
<td>114</td>
<td>9</td>
<td>14664</td>
<td>15372</td>
<td>48</td>
<td></td>
<td>1492</td>
<td>audiodg</td>
</tr>
<tr>
<td>556</td>
<td>62</td>
<td>53928</td>
<td>5368</td>
<td>616</td>
<td>3.17</td>
<td>3408</td>
<td>CCC</td>
</tr>
<tr>
<td>58</td>
<td>8</td>
<td>2960</td>
<td>7068</td>
<td>70</td>
<td>0.19</td>
<td>928</td>
<td>conhost</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>1468</td>
<td>3468</td>
<td>52</td>
<td>0.00</td>
<td>5068</td>
<td>conhost</td>
</tr>
<tr>
<td>784</td>
<td>14</td>
<td>3284</td>
<td>5092</td>
<td>56</td>
<td></td>
<td>416</td>
<td>cssr</td>
</tr>
<tr>
<td>529</td>
<td>27</td>
<td>2928</td>
<td>17260</td>
<td>145</td>
<td></td>
<td>496</td>
<td>cssr</td>
</tr>
<tr>
<td>182</td>
<td>13</td>
<td>8184</td>
<td>11152</td>
<td>96</td>
<td>0.50</td>
<td>2956</td>
<td>DCPSysMgr</td>
</tr>
<tr>
<td>135</td>
<td>11</td>
<td>2880</td>
<td>7552</td>
<td>56</td>
<td></td>
<td>2056</td>
<td>DCPSysMrgSv</td>
</tr>
</tbody>
</table>

... (truncated output)

After a job has been received, that is it—the data is gone, unless you saved it to a variable or you call the `Receive-Job` cmdlet with the `-Keep` switch parameter. The following code attempts to retrieve the information stored from job10, but as shown here, no data returns.

```powershell
PS C:\> Receive-Job -Name job10
PS C:\>
```
What can be confusing about this is that the job still exists, and the Get-Job cmdlet continues to retrieve information about the job. This is shown here.

```
PS C:\> Get-Job -Id 10
Id   Name            PSJobTypeName   State        HasMoreData     Location
--   ----            -------------   -----         -----------     --------
10   Job10           BackgroundJob   Completed     False           localhost
```

As a best practice, use the Remove-Job cmdlet to delete remnants of completed jobs when you are finished using the job object. This will avoid confusion regarding active jobs, completed jobs, and jobs waiting to be processed. After a job has been removed, the Get-Job cmdlet returns an error if you attempt to retrieve information about the job—because it no longer exists. This is illustrated here.

```
PS C:\> Remove-Job -Name job10
PS C:\> Get-Job -Id 10
Get-Job : The command cannot find a job with the job ID 10. Verify the value of the Id parameter and then try the command again.
At line:1 char:1
  + Get-Job -Id 10
  + ~~~~~~~~~~~~~~
    + CategoryInfo : ObjectNotFound: (10:Int32) [Get-Job], PSArgumentException
      + FullyQualifiedErrorId : JobWithSpecifiedSessionNotFound,Microsoft.PowerShell.Commands.GetJobCommand
```

When working with the job cmdlets, I like to give the jobs their own names. A job that returns process objects via the Get-Process cmdlet might be called getProc. A contextual naming scheme works better than trying to keep track of names such as Job1 and Job2. Do not worry about making your job names too long, because you can use wildcard characters to simplify the typing requirement. When you receive a job, make sure you store the returned objects in a variable. This is shown here.

```
PS C:\> Start-Job -Name getProc -ScriptBlock {get-process}
Id   Name            PSJobTypeName   State        HasMoreData     Location
--   ----            -------------   -----         -----------     --------
12   getProc         BackgroundJob   Running       True            localhost
```

```
PS C:\> Get-Job -Name get*
Id   Name            PSJobTypeName   State        HasMoreData     Location
--   ----            -------------   -----         -----------     --------
12   getProc         BackgroundJob   Completed     True            localhost
```

```
PS C:\> $procObj = Receive-Job -Name get*
PS C:\>
```

When you have the returned objects in a variable, you can use the objects with other Windows PowerShell cmdlets. One thing to keep in mind is that the object is deserialized. This is shown on the following page, where I use gm as an alias for the Get-Member cmdlet.
This means that not all the standard members from the `System.Diagnostics.Process` .NET Framework object are available. The default methods are shown here (`gps` is an alias for the `Get-Process` cmdlet, `gm` is an alias for `Get-Member`, and `-m` is enough of the `-MemberType` parameter to distinguish it on the Windows PowerShell console line).

Methods from the deserialized object are shown here, where I use the same command I used previously.
A listing of the cmdlets that use the noun `job` is shown here.

PS C:\> Get-Command -Noun job | select name

Name  
----  
Get-Job  
Receive-Job  
Remove-Job  
Resume-Job  
Start-Job  
Stop-Job  
Suspend-Job  
Wait-Job  

When starting a Windows PowerShell job via the `Start-Job` cmdlet, you can specify a name to hold the returned job object. You can also assign the returned job object in a variable by using a straightforward value assignment.

PS C:\> $rtn = Start-Job -Name net -ScriptBlock {Get-Net6to4Configuration}  
PS C:\> Get-Job -Name net

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>net</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

PS C:\> $rtn

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>net</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

Retrieving the job via the `Receive-Job` cmdlet consumes the data. You cannot come back and retrieve the returned data again. The code shown here illustrates this concept.

PS C:\> Receive-Job $rtn

RunspaceId   : e8ed4ab6-eb88-478c-b2de-5991b5636ef1  
Caption      :  
Description  : 6to4 Configuration  
ElementName  :  
InstanceID   : ActiveStore  
AutoSharing   : 0  
PolicyStore  : ActiveStore  
RelayState   : 0  
ResolutionInterval : 1440  
State        : 0

PS C:\> Receive-Job $rtn  
PS C:\>
The next example illustrates examining the command and cleaning up the job. To find additional information about the code, use the job object stored in the $rtn variable or the Get-Net6to4Configuration job. You might prefer using the job object stored in the $rtn variable, as shown here.

PS C:\> $rtn.Command
Get-Net6to4Configuration

To clean up, first remove the leftover job objects by getting the jobs and removing the jobs. This is shown here.

PS C:\> Get-Job | Remove-Job
PS C:\> Get-Job
PS C:\>

Caution The following example uses the Win32_Product class for illustrative purposes. When the Win32_Product class is queried, it will initiate an MSI consistency check, which can have undesirable effects. When working with this class, use caution.

When you create a new Windows PowerShell job, it runs in the background. There is no indication as the job runs whether it ends in an error or it’s successful. Indeed, you do not have any way to tell when the job even completes, other than to use the Get-Job cmdlet several times to find out when the job state changes from running to completed. For many jobs, this might be perfectly acceptable. In fact, it might even be preferable, if you want to regain control of the Windows PowerShell console as soon as the job begins executing. On other occasions, you might want to be notified when the Windows PowerShell job completes. To accomplish this, you can use the Wait-Job cmdlet. You need to give the Wait-Job cmdlet either a job name or a job ID. After you have done this, the Windows PowerShell console will pause until the job completes. The job, with its completed status, displays on the console. You can then use the Receive-Job cmdlet to receive the deserialized objects and store them in a variable (cn is a parameter alias for the -ComputerName parameter used in the Get-WmiObject command). The command shown here starts a job to receive software products installed on a remote server named hyperv1. It impersonates the currently logged-on user and stores the returned object in a variable named $rtn.

PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Job22</td>
<td>BackgroundJob</td>
<td>Running</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

PS C:\> Wait-Job -id 22

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Job22</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

PS C:\> $prod = Receive-Job -id 22
PS C:\> $prod.Count
2
In a newly opened Windows PowerShell console, the `Start-Job` cmdlet is used to start a new job. The returned job object is stored in the `$rtn` variable. You can pipeline the job object contained in the `$rtn` variable to the `Stop-Job` cmdlet to stop the execution of the job. If you try to use the job object in the `$rtn` variable directly to get job information, an error will be generated. This is shown here.

```powershell
PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn | Stop-Job
PS C:\> Get-Job $rtn
Get-Job : The command cannot find the job because the job name
System.Management.Automation.PSRemotingJob was not found. Verify the value of the
Name parameter, and then try the command again.
At line:1 char:1
+ Get-Job $rtn
+ ~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (System.Management.Automation.PSRemotingJob: String) [Get-Job], PSArgumentException
    + FullyQualifiedErrorId : JobWithSpecifiedNameNotFound,Microsoft.PowerShell.Commands.GetJobCommand
```

You can pipeline the job object to the `Get-Job` cmdlet and find that the job is in a stopped state. Use the `Receive-Job` cmdlet to receive the job information, and the `count` property to determine how many software products are included in the variable, as shown here.

```powershell
PS C:\> $rtn | Get-Job
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
2      Job2            BackgroundJob   Stopped       False           localhost

PS C:\> $products = Receive-Job -Id 2
PS C:\> $products.count
0
```

In the preceding list you can tell that no software packages were enumerated. This is because the `Get-WmiObject` command to retrieve information from the `Win32_Product` class did not have time to finish.

If you want to keep the data from your job so that you can use it again later, and you do not want to bother storing it in an intermediate variable, use the `-Keep` switch parameter. In the command that follows, the `Get-NetAdapter` cmdlet is used to return network adapter information.

```powershell
PS C:\> Start-Job -ScriptBlock {Get-NetAdapter}
Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
4      Job4            BackgroundJob   Running       True            localhost
```

From the Library of Todd Schultz
When checking on the status of a background job and monitoring a job you just created, use the -Newest parameter instead of typing a job number, because it is easier to remember. This technique is shown here.

```
PS C:\> Get-Job -Newest 1
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Job4</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

Now, to retrieve the information from the job and keep the information available, use the -Keep switch parameter, as illustrated here.

```
PS C:\> Receive-Job -Id 4 -Keep
```

```
ifAlias                                          : Ethernet
InterfaceAlias                                   : Ethernet
ifIndex                                          : 4
ifDesc                                           : Microsoft Hyper-V Network Adapter
ifName                                           : Ethernet_32768
DriverVersion                                    : 10.0.10224.0
LinkLayerAddress                                 : 00-15-5D-00-2A-1E
MacAddress                                       : 00-15-5D-00-2A-1E
LinkSpeed                                        : 10 Gbps
MediaType                                        : 802.3
PhysicalMediaType                                : Unspecified
AdminStatus                                      : Up
MediaConnectionState                             : Connected
DriverInformation                                : Driver Date 2006-06-21 Version
                                                   : 10.0.10224.0 NDIS 6.50
DriverFileName                                   : netvsc.sys
NdisVersion                                      : 6.50
ifOperStatus                                     : Up
RunspaceId                                       : d5fc4fb6-4db2-46f4-9d38-a79f1c0e0
Caption                                          : 
Description                                      : 
ElementName                                      : 
InstanceID                                       : {A03CCF8C-6D91-49C0-ACBD-B900FC27EAC1}
CommunicationStatus                             : 
DetailedStatus                                   : 
HealthState                                      : 
InstallDate                                      : 
Name                                            : Ethernet
OperatingStatus                                  : 
OperationalStatus                                : 
PrimaryStatus                                    : 
Status                                           : 
```
StatusDescriptions : 
AvailableRequestedStates : 
EnabledDefault : 2
EnabledState : 5
OtherEnabledState : 
RequestedState : 12
TimeOfLastStateChange : 
TransitioningToState : 12
AdditionalAvailability : 
Availability : 
CreationClassName : MSFT_NetAdapter
DeviceID : {A03CCF8C-6D91-49C0-ACBD-B900FC27EAC1}

ErrorCleared : 
ErrorDescription : 
IdentifyingDescriptions : 
LastErrorCode : 
MaxQuiesceTime : 
OtherIdentifyingInfo : 
PowerManagementCapabilities : 
PowerManagementSupported : 
PowerOnHours : 
StatusInfo : 
SystemCreationClassName : CIM_NetworkPort
SystemName : c10.NWTraders.com
TotalPowerOnHours : 
MaxSpeed : 
OtherPortType : 
PortType : 
RequestedSpeed : 
Speed : 10000000000
UsageRestriction : 
ActiveMaximumTransmissionUnit : 1500
AutoSense : 
FullDuplex : 
LinkTechnology : 
NetworkAddresses : {00155D002A1E}
OtherLinkTechnology : 
OtherNetworkPortType : 
PermanentAddress : 00155D002A1E
PortNumber : 0
SupportedMaximumTransmissionUnit : 
AdminLocked : False 
ComponentID : VMBUS\{f8615163-df3e-46c5-913f-f2d2f965ed0e}
ConnectorPresent : True 
DeviceName : \\Device\{A03CCF8C-6D91-49C0-ACBD-B900FC27EAC1}

DeviceWakeUpEnable : False 
DriverDate : 2006-06-21
DriverDateData : 127953216000000000
DriverDescription : Microsoft Hyper-V Network Adapter
DriverMajorNdisVersion : 6
DriverMinorNdisVersion : 50
DriverName : \SystemRoot\System32\drivers\ntvsc.sys
DriverProvider : Microsoft
You can continue to work directly with the output in a normal Windows PowerShell fashion, as follows.

PS C:\> Receive-Job -Id 4 -Keep | select name

name
----
Ethernet

PS C:\> Receive-Job -Id 4 -Keep | select transmitlinksp*

TransmitLinkSpeed
-----------------
10000000000
Using Windows PowerShell remoting and jobs: Step-by-step exercises

In this exercise, you will practice using Windows PowerShell remoting to run remote commands. For the purpose of this exercise, you can use your local computer. First, you will open the Windows PowerShell console, supply alternate credentials, create a Windows PowerShell remote session, and run various commands. Next, you will create and receive Windows PowerShell jobs.

**Supplying alternate credentials for remote Windows PowerShell sessions**

1. Log on to your computer with a user account that does not have administrator rights.

2. Open the Windows PowerShell console.

3. Notice the Windows PowerShell console prompt. An example of such a prompt is shown here.

   ```
   PS C:\Users\ed.nwtraders>
   ```

4. Use a variable named `$cred` to store the results of using the `Get-Credential` cmdlet. Specify administrator credentials to store in the `$cred` variable. An example of such a command is shown here.

   ```
   $cred = Get-Credential nwtraders\administrator
   ```

5. Use the `Enter-PSSession` cmdlet to open a remote Windows PowerShell console session. Use the credentials stored in the `$cred` variable, and use `localhost` as the name of the remote computer. An example of this command is shown here.

   ```
   Enter-PSSession -ComputerName localhost -Credential $cred
   ```

6. Notice how the Windows PowerShell console prompt changes to include the name of the remote computer and also changes the working directory. An example of a changed prompt is shown here.

   ```
   [localhost]: PS C:\Users\administrator\Documents>
   ```

7. Use the `whoami` command to verify the current context. The results of the command are shown here.

   ```
   [localhost]: PS C:\Users\administrator\Documents> whoami
   nwtraders\administrator
   ```

8. Use the `exit` command to exit the remote session. Use the `whoami` command to verify that the user context has changed.
9. Use WMI to retrieve the BIOS information on the local computer. Use the alternate credentials stored in the $cred variable. This command is shown here.

```powershell
gwmi -Class win32_bios -cn localhost -Credential $cred
```

The previous command fails and produces the following error. This error comes from WMI and states that you are not permitted to use alternate credentials for a local WMI connection.

```
gwmi : User credentials cannot be used for local connections
At line:1 char:1
+ gwmi -Class win32_bios -cn localhost -Credential $cred
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], ManagementException
  + FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

10. Put the WMI command into the -ScriptBlock parameter for Invoke-Command. Specify the local computer as the value for -ComputerName, and use the credentials stored in the $cred variable. The command is shown here (using -script as a shortened version of -ScriptBlock).

```
Invoke-Command -cn localhost -script {gwmi -Class win32_bios} -cred $cred
```

11. Press the Up Arrow key to retrieve the previous command, and erase the credential parameter. The revised command is shown here.

```
Invoke-Command -cn localhost -script {gwmi -Class win32_bios}
```

When you run the command, it generates the error shown here because a normal user does not have remote access by default (if you have admin rights, the command works).

```
[localhost] Connecting to remote server localhost failed with the following error message : Access is denied. For more information, see the about_Remote_Troubleshooting Help topic.
  + CategoryInfo          : OpenError: (localhost:String) [], PSRemotingTransportException
  + FullyQualifiedErrorId : AccessDenied,PSSessionStateBroken
```

12. Create an array of computer names. Store the computer names in a variable named $cn. Use the array shown here.

```
$cn = $env:COMPUTERNAME,"localhost","127.0.0.1"
```

13. Use Invoke-Command to run the WMI command on all three computers at the same time. The command is shown here.

```
Invoke-Command -cn $cn -script {gwmi -Class win32_bios} -cred $cred
```

This concludes this step-by-step exercise.

In the following exercise, you will create and receive Windows PowerShell jobs.
Creating and receiving jobs

1. Open the Windows PowerShell console as a non-elevated user.

2. Start a job named Get-Process that uses a -ScriptBlock parameter that calls the Get-Process cmdlet (gps is an alias for Get-Process). The command is shown here.

   ```powershell
   Start-Job -Name gps -ScriptBlock {gps}
   ```

3. Examine the output from starting the job. It lists the name, state, and other information about the job. Sample output is shown here.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>gps</td>
<td>BackgroundJob</td>
<td>Running</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

4. Use the Get-Process cmdlet to determine whether the job has completed. The command is shown here.

   ```powershell
   Get-Job gps
   ```

5. Examine the output from the previous command. The state reports completed when the job has completed. If data is available, the HasMoreData property reports True. Sample output is shown here.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>gps</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>True</td>
<td>localhost</td>
</tr>
</tbody>
</table>

6. Receive the results from the job. To do this, use the Receive-Job cmdlet, as shown here.

   ```powershell
   Receive-Job gps
   ```

7. Press the Up Arrow key to retrieve the Get-Job command. Run it. Note that the HasMoreData property now reports False, as shown here.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>PSJobTypeName</th>
<th>State</th>
<th>HasMoreData</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>gps</td>
<td>BackgroundJob</td>
<td>Completed</td>
<td>False</td>
<td>localhost</td>
</tr>
</tbody>
</table>

8. Create a new job with the same name as the previous job: gps. This time, change the -Script-Block parameter value to gsv (the alias for Get-Service). The command is shown here.

   ```powershell
   Start-Job -Name gps -ScriptBlock {gsv}
   ```
9. Now use the Get-Job cmdlet to retrieve the job with the namegps. Note that the command retrieves both jobs, as shown here.

   Get-Job -name gps

   Id     Name            PSJobTypeName   State         HasMoreData     Location
   --     ----            -------------   -----         -----------     --------
   9      gps             BackgroundJob   Completed     False           localhost
   11     gps             BackgroundJob   Completed     True            localhost

10. Use the Receive-Job cmdlet to retrieve the job ID associated with your new job. This time, use the -Keep switch parameter, as shown here.

    Receive-Job -Id 11 -keep

11. Use the Get-Job cmdlet to retrieve your job. Note that the HasMoreData property still reports True because you’re using the -Keep switch parameter.

    This concludes this exercise.

Chapter 4 quick reference

<table>
<thead>
<tr>
<th>To</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work interactively on a remote system</td>
<td>Use the Enter-PSSession cmdlet to create a remote session.</td>
</tr>
<tr>
<td>Configure Windows PowerShell remoting</td>
<td>Use the Enable-PSRemoting cmdlet.</td>
</tr>
<tr>
<td>Run a command on a remote system</td>
<td>Use the Invoke-Command cmdlet and specify the command by using the -ScriptBlock parameter.</td>
</tr>
<tr>
<td>Run a command as a job</td>
<td>Use the Start-Job cmdlet to execute the command.</td>
</tr>
<tr>
<td>Check on the progress of a job</td>
<td>Use the Get-Job cmdlet and specify either the job ID or the job name.</td>
</tr>
<tr>
<td>Check on the progress of the newest job</td>
<td>Use the Get-Job cmdlet and specify the -Newest parameter, and supply the number of newest jobs to monitor.</td>
</tr>
<tr>
<td>Retrieve the results from a job</td>
<td>Use the Receive-Job cmdlet and specify the job ID.</td>
</tr>
</tbody>
</table>