

# POWERSHELL 2.0 – ONE CMDLET AT A TIME

Jonathan Medd

## Introduction

Back in November 2009 I decided it was time to really crack on getting to grips with PowerShell version 2.0. The full release had been out for a couple of weeks, since it had shipped as part of Windows 7 / Windows Server 2008 R2 after three previews of what might be via Community Technology Previews.

Whilst there are a number of big new features in PowerShell 2.0 I decided to start by taking a look at some of the new cmdlets and making a blog post for each one as I went, which would force me to learn properly and hopefully make a decent community contribution at the same time.

After starting the series with the initial intention of covering maybe at most 20 cmdlets I was blackmailed encouraged to keep going and ended up covering over 100+ cmdlets. I did most of this during my lunchtimes at work, found it a great way to get to grips with the new functionality and managed to keep it going through to June 2010.

Whilst about halfway through the series I started to think about compiling all of the blog posts into one handy reference document and make it available for download from my blog. Since this makes it slightly more formal, rather than the happy-go-lucky nature of a blog post I decided to get some real experts to review the content to make sure it was decent and accurate so three PowerShell MVPs kindly spent their own time reviewing it for me.

A big thank you goes out to these gentlemen; Thomas Lee (<http://twitter.com/doctordns>), Richard Siddaway (<http://msmvps.com/blogs/RichardSiddaway/Default.aspx>) and Aleksandar Nikolic (<http://twitter.com/alexandair>). Thank you for all your feedback and comments which have been incorporated into this document.

Also thanks to those who have followed the blog series since I started it, left comments, re-tweeted each one and generally encouraged me to keep going.

I hope you find this consolidated series useful. If you do then I ask that you consider making a small donation to a UK based charity that help the parents of children born with Tracheo-Oesophageal Fistula (TOF) and Oesophageal Atresia (OA). You can find out more about this charity through their website <http://www.tofs.org.uk/index.php> and can make a donation here <http://www.justgiving.com/tofs/donate>. I know from personal experience what a great job they do.

Thanks!

Jonathan

<http://jonathanmedd.net>

<http://twitter.com/jonathanmedd>

September 2010

## Contents

Introduction .....	1
#1 Get-Random .....	6
#2 Send-MailMessage .....	7
#3 Get-Counter .....	8
#4 Out-GridView .....	9
#5 Get-HotFix .....	11
#6 Test-Connection .....	12
#7 Reset-ComputerMachinePassword .....	13
#8 Get-Module .....	14
#9 Checkpoint-Computer .....	15
#10 Restart-Computer .....	16
#11 Add-Computer .....	17
#12 Write-EventLog .....	19
#13 Clear-EventLog .....	20
#14 Start-Process .....	21
#15 Start-Job .....	22
#16 Get-Job .....	24
#17 Receive-Job .....	26
#18 Remove-Job .....	27
#19 Stop-Job .....	29
#20 Wait-Job .....	31
#21 Select-XML .....	33
#22 Enable-ComputerRestore .....	35
#23 Disable-ComputerRestore .....	36
#24 Get-ComputerRestorePoint .....	37
#25 Restore-Computer .....	38
#26 New-WebServiceProxy .....	39
#27 Test-ComputerSecureChannel .....	41
#28 Export-Counter .....	43
#29 Import-Counter .....	44
#30 Enable-PSRemoting .....	45
#31 Enter-PSSession .....	47
#32 Exit-PSSession .....	48

#33 New-PSSession .....	49
#34 Invoke-Command .....	50
#35 New-PSSessionOption .....	52
#36 Get-PSSession .....	54
#37 Remove-PSSession .....	56
#38 Get-PSSessionConfiguration .....	58
#39 Register-PSSessionConfiguration .....	59
#40 Set-PSSessionConfiguration .....	61
#41 Disable-PSSessionConfiguration .....	63
#42 Enable-PSSessionConfiguration .....	65
#43 Unregister-PSSessionConfiguration .....	67
#44 Set-WSManQuickConfig .....	69
#45 Connect-WSMan .....	71
#46 Test-WSMan .....	74
#47 Invoke-WSManAction .....	76
#48 Get-WSManInstance .....	78
#49 New-WSManInstance .....	80
#50 Set-WSManInstance .....	85
#51 Remove-WSManInstance .....	87
#52 New-WSManSessionOption .....	89
#53 Enable-WSManCredSSP .....	91
#54 Get-WSManCredSSP .....	93
#55 Disable-WSManCredSSP .....	95
#56 Disconnect-WSMan .....	97
#57 Import-PSSession .....	98
#58 Export-PSSession .....	100
#59 Set-PSBreakpoint .....	102
#60 Get-PSBreakpoint .....	104
#61 Disable-PSBreakpoint .....	105
#62 Enable-PSBreakpoint .....	106
#63 Remove-PSBreakpoint .....	107
#64 Clear-History .....	109
#65 New-EventLog .....	111
#66 Limit-EventLog .....	113

#67 Remove-EventLog .....	115
#68 Show-EventLog.....	117
#69 Get-WinEvent.....	119
#70 Import-Module.....	121
#71 New-Module .....	123
#72 Export-ModuleMember .....	125
#73 New-ModuleManifest.....	127
#74 Test-ModuleManifest .....	130
#75 Remove-Module .....	132
#76 Stop-Computer.....	133
#77 Remove-Computer.....	134
#78 Start-Transaction.....	135
#79 Complete-Transaction.....	137
#80 Get-Transaction.....	139
#81 Undo-Transaction.....	141
#82 Use-Transaction .....	143
#83 ConvertTo-CSV .....	145
#84 ConvertFrom-CSV.....	146
#85 ConvertFrom-StringData.....	147
#86 ConvertTo-XML .....	148
#87 Get-FormatData .....	150
#88 Export-FormatData .....	152
#89 Invoke-WmiMethod .....	154
#90 Remove-WmiObject.....	156
#91 Set-WmiInstance .....	157
#92 Register-WmiEvent .....	159
#93 Register-ObjectEvent .....	161
#94 Get-EventSubscriber .....	163
#95 Register-EngineEvent .....	165
#96 New-Event.....	167
#97 Get-Event .....	169
#98 Wait-Event .....	171
#99 Unregister-Event .....	173
#100 Remove-Event.....	174

#101 Wait-Process .....	176
#102 Disable-PSRemoting .....	178
#103 Update-List.....	180
#104 Trace-Command.....	181
#105 Set-StrictMode .....	183
#106 Import-LocalizedData.....	185
#107 Add-Type .....	187

## #1 Get-Random

### Get-Random

#### What can I do with it?

With Get-Random you can either generate a random number, or randomly select objects from a collection.

#### Examples:

Generate a random number between 1 and 100.

```
Get-Random -Minimum 1 -Maximum 101
```

Select a random object from a collection

```
$users = 'Rod', 'Jane', 'Freddy'  
Get-Random $users
```

Select a random Windows Service

```
Get-Service | Get-Random
```

#### How could I have done this in PowerShell 1.0?

Generate a random number between 1 and 100 using .NET.

```
$randomnumber = New-Object System.Random  
$randomnumber.next(1,101)
```

Select a random object from a collection

```
$users = 'Rod', 'Jane', 'Freddy'  
$randomnumber = New-Object System.Random  
$i = $randomnumber.next(0,$users.length)  
$users[$i]
```

How did I decide to begin this series with Get-Random?

```
Get-Command | Get-Random
```

;-)

## #2 Send-MailMessage

### Send-MailMessage

#### What can I do with it?

Send an email message using a specific SMTP server, from within a script or at the command line.

#### Example:

```
Send-MailMessage -To "Joe Bloggs <joe.bloggs@test.local>" -From "Jane Smith <jane.smith@test.local>"  
-Subject "Reporting Document" -Body "Here's the document you wanted" -  
Attachments "C:\Report.doc"  
-SmtpServer smtp.test.local
```

#### How could I have done this in PowerShell 1.0?

You could have used the .NET System.Net.Mail class

```
Function Send-MailMessage ()  
{param ($Sender,$Recipient,$Attachment)  
  
$smtpServer = "smtp.test.local"  
  
$msg = New-Object System.Net.Mail.MailMessage  
$att = New-Object System.Net.Mail.Attachment($Attachment)  
$smtp = New-Object System.Net.Mail.SmtpClient($smtpServer)  
  
$msg.From = "$Sender"  
$msg.To.Add("$Recipient")  
$msg.Subject = "Reporting Document"  
$msg.Body = "Here's the document you wanted."  
$msg.Attachments.Add($att)  
  
$smtp.Send($msg)  
  
$att.Dispose();  
}  
  
Send-MailMessage 'jane.smith@test.local' 'joe.bloggs@test.local' 'C:\Report.doc'
```

## #3 Get-Counter

[Get-Counter](#).

### What can I do with it?

Collect real-time performance counter data directly from local or remote computers.

#### Examples:

Create a list of performance counters available to query in the Memory counter

```
(Get-Counter -ListSet memory) .paths
```

**Tip:** To find a list of available top-level counters for which you could substitute in for **memory** in the above example you could type this set of commands:

```
Get-Counter -ListSet * | Sort-Object counterSetName | Format-Table  
counterSetName
```

To retrieve the current Memory Pool Paged Bytes on the remote computer Server1

```
Get-Counter -Counter '\Memory\Pool Paged Bytes' -ComputerName Server1
```

**Tip:** You can run multiple samples using the **-MaxSamples** parameter

```
Get-Counter -Counter '\Memory\Pool Paged Bytes' -ComputerName Server1 -  
MaxSamples 5
```

### How could I have done this in PowerShell 1.0?

You could use the **Get-WmiObject** cmdlet and the **Win32\_PerfFormattedData** class to look at performance data for a remote computer. For example:

```
(Get-WmiObject Win32_PerfFormattedData_PerfOS_Memory -ComputerName  
Server1) .PoolPagedBytes
```

You could also use .NET and the **System.Diagnostics.PerformanceCounter** class to view performance data

```
$data = New-Object System.Diagnostics.PerformanceCounter  
$data.CategoryName = "Memory"  
$data.CounterName = "Pool Paged Bytes"  
$data.NextValue()
```

Thanks to [MoiW](#) for the .NET info.

### Related Cmdlets

[Export-Counter](#)

[Import-Counter](#)

## #4 Out-GridView

[Out-GridView](#).

### What can I do with it?

View the output from a command in an interactive grid window.

### Any special requirements?

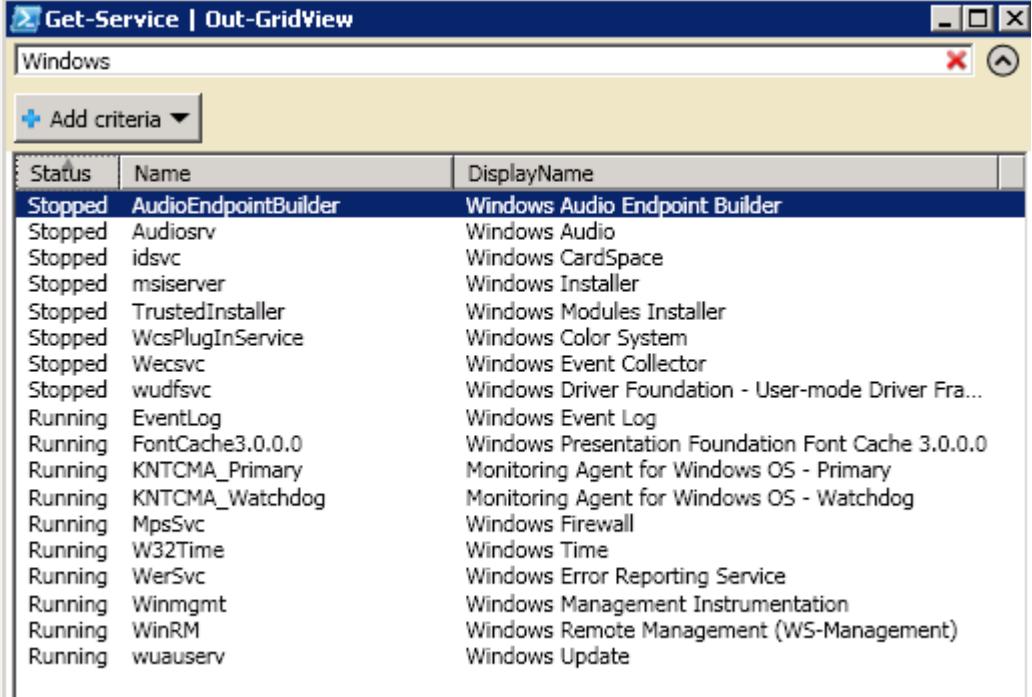
Whilst PowerShell 2.0 itself requires .NET Framework 2.0 with Service Pack 1, this particular cmdlet requires .NET Framework 3.5 Service Pack 1.

### Examples:

Create an interactive grid view of the list of services running on the machine.

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

The resulting output with a filter of **Windows** and sorted by **Status** gives you an idea for what you can use this for:



Status	Name	DisplayName
Stopped	AudioEndpointBuilder	Windows Audio Endpoint Builder
Stopped	Audiosrv	Windows Audio
Stopped	idsvc	Windows CardSpace
Stopped	msiserver	Windows Installer
Stopped	TrustedInstaller	Windows Modules Installer
Stopped	WcsPlugInService	Windows Color System
Stopped	Webservice	Windows Event Collector
Stopped	wudfsvc	Windows Driver Foundation - User-mode Driver Fra...
Running	EventLog	Windows Event Log
Running	FontCache3.0.0.0	Windows Presentation Foundation Font Cache 3.0.0.0
Running	KNTCMA_Primary	Monitoring Agent for Windows OS - Primary
Running	KNTCMA_Watchdog	Monitoring Agent for Windows OS - Watchdog
Running	MpsSvc	Windows Firewall
Running	W32Time	Windows Time
Running	WerSvc	Windows Error Reporting Service
Running	Winmgmt	Windows Management Instrumentation
Running	WinRM	Windows Remote Management (WS-Management)
Running	wuauserv	Windows Update

Create an interactive grid view of the **System** log with the latest 5 entries, selecting only the **Source** and **Message** properties and displaying the Output with a custom title.

```
Get-Eventlog -LogName System -Newest 5 | Select-Object Source,Message | Out-GridView -Title 'System Log'
```

System Log	
Filter	
Add criteria ▾	
Source	Message
Microsoft-Windows-GroupPolicy	The Group Policy settings for the computer were processed successfully. New settings from 4
EventLog	The system uptime is 960846 seconds.
Microsoft-Windows-GroupPolicy	The Group Policy settings for the computer were processed successfully. New settings from 4
Service Control Manager	The description for Event ID '1073748860' in Source 'Service Control Manager' cannot be fou
Microsoft-Windows-GroupPolicy	The Group Policy settings for the computer were processed successfully. New settings from 4

**Tip:**

I don't recommend using aliases within a script, but this is the kind of cmdlet you are most likely to use when working at the command line so the alias for Out-GridView, `ogv`, could come in very handy.

**How could I have done this in PowerShell 1.0?**

The closest you could probably get would be Export-CSV to give you the data in a CSV file, which could then be manipulated in a similar fashion to Out-GridView using Excel.

```
Get-Service | Export-Csv C:\Scripts\Services.csv -NoTypeInformation
```

The CSV file can be opened in Excel using PowerShell:

```
Invoke-Item C:\Scripts\Services.csv
```

## #5 Get-HotFix

[Get-HotFix.](#)

### What can I do with it?

Retrieve hotfixes installed on a local or remote computer

#### Example:

Retrieve a list of hotfixes installed on Server1 which contain **Security** in their description.  
Display the **Description**, **HotfixID** and **Caption** properties.

```
Get-HotFix -Description Security* -ComputerName Server01  
| Select-Object Description,HotfixID,Caption
```

### How could I have done this in PowerShell 1.0?

You could have used **Get-WmiObject** with the **Win32\_QuickFixEngineering** class.

```
Get-WmiObject -Class Win32_QuickFixEngineering -Filter "Description LIKE  
'Security%'"  
-ComputerName Server01 | Select-Object Description,HotfixID,Caption
```

Funnily enough **Get-HotFix** and **Get-WmiObject -Class Win32\_QuickFixEngineering** look pretty similar when you pipe them through to **Get-Member** ;-)

## #6 Test-Connection

[Test-Connection](#).

**What can I do with it?**

Send a ping to one or more computers

**Examples:**

Send a ping to Server01

```
Test-Connection -ComputerName Server01
```

If the result of a ping to Server01 is successful then copy a text file to a file share on that server

```
If (Test-Connection -ComputerName Server01 -Quiet)
{Copy-Item C:\Document.txt "\\\Server01\Fileshare"}
```

**How could I have done this in PowerShell 1.0?**

You could have used **Get-WmiObject** with the **Win32\_PingStatus** class.

```
Get-WmiObject Win32_PingStatus -Filter "Address='Server01'"
```

Funnily enough **Test-Connection** and **Get-WmiObject -Class Win32\_PingStatus** look pretty similar when you pipe them through to **Get-Member** 😊

You could also have used the .NET **System.Net.NetworkInformation.Ping** class

```
$ping = New-Object System.Net.NetworkInformation.Ping
$ping.send('Server01')
```

**Related Cmdlets**

[Restart-Computer](#)

[Stop-Computer](#)

## #7 Reset-ComputerMachinePassword

### Reset-ComputerMachinePassword

#### What can I do with it?

Reset the computer account password for a machine.

#### Examples:

Reset the computer account password for the current local machine. It's as simple as that!

#### `Reset-ComputerMachinePassword`

To do the same for a remote machine you will need to use Invoke-Command to run the command on the remote machine.

```
Invoke-Command -ComputerName Server01  
-ScriptBlock {Reset-ComputerMachinePassword}
```

#### How could I have done this in PowerShell 1.0?

You could have done the following.

```
[ADSI]$computer = "WinNT://WINDOWS2000/computername$"  
$computer.SetPassword("computername$")
```

More commonly you might have used the netdom command line tool to do this.

```
netdom reset 'machinename' /domain:'domainname'
```

Or you might have used Active Directory Users and Computers GUI tool, right-clicked the computer account in question and chosen **Reset Account**.

## #8 Get-Module

### [Get-Module](#)

#### **What can I do with it?**

PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. There are some great videos below by Bruce Payette and Osama Sajid from the PowerShell team both introducing and demonstrating how to use modules:  
[\(Thanks Shay\)](#)

Episode [one](#) introduces Modules and discusses comparisons with Cmdlets.

Episode [two](#) demonstrates how to use Modules.

Episode [three](#) illustrates how to develop script and binary Modules

#### **Example:**

Retrieve all the modules on the current system which could be imported into the current session

`Get-Module -ListAvailable`

#### **How could I have done this in PowerShell 1.0?**

You could have used the Get-PSSnapin cmdlet to see which snap-ins were available to use.  
To see snap-ins available in the current session:

`Get-PSSnapin`

To see snap-ins available to add to the current session use the Registered parameter:

`Get-PSSnapin -Registered`

#### **Related Cmdlets**

[Import-Module](#)

[New-Module](#)

[Remove-Module](#)

## #9 Checkpoint-Computer

[Checkpoint-Computer](#).

### What can I do with it?

Create a system restore point on XP, Vista or Windows 7 systems.

#### Example:

Create a system restore point called Pre-RegistryChange

```
Checkpoint-Computer -Description "Pre-RegistryChange"
```

### How could I have done this in PowerShell 1.0?

You could have used the **SystemRestore** [WMI class](#) and the **CreateRestorePoint** method

```
$SystemRestore = [wmiclass]"\\.\root\default:systemrestore"  
$SystemRestore.CreateRestorePoint("Pre-RegistryChange", 0, 100)
```

#### Related Cmdlets

[Add-Computer](#)

[Get-ComputerRestorePoint](#)

[Remove-Computer](#)

[Restart-Computer](#)

[Restore-Computer](#)

[Stop-Computer](#)

## #10 Restart-Computer

[Restart-Computer.](#)

### What can I do with it?

Restart a local or remote computer

#### Example:

Immediately restart the computer Server01.

```
Restart-Computer -ComputerName Server01 -Force
```

### How could I have done this in PowerShell 1.0?

You could have used the **Win32\_OperatingSystem** [WMI Class](#) and the **Win32Shutdown** method.

```
(Get-WmiObject -Class Win32_OperatingSystem  
-ComputerName Server01).Win32Shutdown(2)
```

Alternatively the Systinternals tool [PSShutdown](#) could be used to restart a local or remote computer.

### Related Cmdlets

[Add-Computer](#)

[Checkpoint-Computer](#)

[Remove-Computer](#)

[Restore-Computer](#)

[Stop-Computer](#)

## #11 Add-Computer

[Add-Computer](#).

### What can I do with it?

Join a local computer to a domain or workgroup

#### Example:

Join the current computer to the Test domain, place the computer account in the Servers OU and use the [Restart-Computer](#) cmdlet to reboot the computer to complete the process.

```
Add-Computer -DomainName Test  
-OUPath 'OU=Servers,DC=test,DC=local'; Restart-Computer
```

### How could I have done this in PowerShell 1.0?

You could have used the [Win32\\_ComputerSystem WMI Class](#) and the [JoinDomainOrWorkGroup method](#).

This [script](#) from the Poshcode script repository illustrates how you might use this method to join a computer to a domain.

```
function Set-Domain {  
    param([switch]$help,  
          [string]$domain=$([read-host] "Please specify the domain to  
join"),  
          [System.Management.Automation.PSCredential]$credential = $(Get-  
Credential))  
  
    $usage = "`$cred = get-credential `n"  
    $usage += "Set-AvaDomain -domain corp.avanade.org -credential `$cred`n"  
    if ($help) {Write-Host $usage;exit}  
  
    $username = $credential.GetNetworkCredential().UserName  
    $password = $credential.GetNetworkCredential().Password  
    $computer = Get-WmiObject Win32_ComputerSystem  
    $computer.JoinDomainOrWorkGroup($domain,$password,$username,$null,  
3)  
}
```

Alternatively you could use the command line tool netdom to join a computer to a domain:

```
NETDOM /Domain:Test /user:adminuser /password:apassword MEMBER  
Server01 /JOINDOMAIN
```

### Related Cmdlets

[Checkpoint-Computer](#)

[Remove-Computer](#)

[Restart-Computer](#)

[Restore-Computer](#)

[Stop-Computer](#)

## #12 Write-EventLog

[Write-EventLog](#).

### What can I do with it?

Write an event in a Windows Event Log on a local or remote machine.

#### Example:

Write an Error event into the Application log on Server01 with source CustomApp1, EventID 8750 and Error Message.

```
Write-EventLog -ComputerName Server01 -LogName Application  
-Source CustomApp1 -EventID 8750 -EntryType Error  
-Message "CustomApp1 has experienced Error 9875"
```

### How could I have done this in PowerShell 1.0?

You could have used the .NET [System.Diagnostics.EventLog](#) class. Richard Siddaway has put together a [great function](#) which uses this class to make it easy to write to the Event Log using PowerShell 1.0.

```
function Write-EventLog  
{  
param([string]$msg = "Default Message", [string]$type="Information")  
$log = New-Object System.Diagnostics.EventLog  
$log.set_log("Application")  
$log.set_source("CustomApp1")  
$log.WriteEntry($msg,$type)  
}
```

You can then use the function like this

```
Write-EventLog "CustomApp1 has started" "Information"
```

### Related Cmdlets

[Clear-EventLog](#)

[Limit-EventLog](#)

[New-EventLog](#)

[Remove-EventLog](#)

[Show-EventLog](#)

[Get-WinEvent](#)

## #13 Clear-EventLog

[Clear-EventLog](#).

### What can I do with it?

Clear the Event Log on a local or remote computer.

#### Example:

Clear the Application Event Log on the remote computer Server01

```
Clear-EventLog -LogName Application -ComputerName Server01
```

### How could I have done this in PowerShell 1.0?

You could have used the **Get-EventLog** cmdlet and the **Clear** method of the System.Diagnostics.EventLog object it generates. (Note: this would only work on a local computer)

```
$ApplicationLog = Get-EventLog -list | Where-Object {$_ .log -eq "Application"}  
$ApplicationLog.Clear()
```

### Related Cmdlets

[Limit-EventLog](#)

[New-EventLog](#)

[Remove-EventLog](#)

[Show-EventLog](#)

[Write-EventLog](#)

[Get-WinEvent](#)

## #14 Start-Process

### Start-Process

#### **What can I do with it?**

Start a process on the local computer.

#### **Examples:**

Start an instance of Notepad

```
Start-Process Notepad
```

Open the file Test.txt using its associated application Notepad

```
Start-Process C:\Scripts\Test.txt
```

#### **How could I have done this in PowerShell 1.0?**

You could have used the .NET [System.Diagnostics.Process](#) class and the [Start](#) method.

```
[System.Diagnostics.Process]::Start("Notepad")
```

and to open a specific file with Notepad

```
[System.Diagnostics.Process]::Start("Notepad", "C:\Scripts\Test.txt")
```

Alternatively you could have used WMI and this option would also give you the ability to start a process on a remote computer.

```
([WMICLASS]"\\Server01\ROOT\CIMV2:win32_process").Create("Notepad")
```

#### **Related Cmdlets**

[Start-Service](#)

[Get-Process](#)

[Stop-Process](#)

[Wait-Process](#)

[Debug-Process](#)

## #15 Start-Job

### Start-Job

#### What can I do with it?

Start a background job on the local computer. This allows you to take back your console session whilst you wait for the job to complete.

#### Examples:

Start a background job to run Get-Service on the local computer.

```
Start-Job -ScriptBlock {Get-Service}
```

This will display the status of this job in your current session and allow you to continue working in the session - then retrieve the results at a later time.

Administrator: Windows PowerShell					
PS C:\> Start-Job -ScriptBlock {Get-Service}					
Id	Name	State	HasMoreData	Location	Command
1	Job1	Running	True	localhost	Get-Service
PS C:\>					

You could also start a background job with a script, not just a scriptblock or a command.

```
Start-Job -FilePath .\Test.ps1
```

Administrator: Windows PowerShell					
PS C:\> Start-Job -FilePath .\Test.ps1					
Id	Name	State	HasMoreData	Location	Command
11	Job11	Running	True	localhost	Write-Host "Hello World"
PS C:\>					

To start a background job on a remote computer use the **-AsJob** parameter available on a number of cmdlets.

(Tip: to find out which cmdlets have the -AsJob parameter use Get-Help to give you a list

```
Get-Help * -Parameter AsJob  
)
```

So to start a job to find services on the remote computer Server1

```
Get-WmiObject Win32_Service -ComputerName Server1 -AsJob
```



```
Administrator: Windows PowerShell
PS C:\> Get-WmiObject Win32_Service -ComputerName Server1 -AsJob
Id          Name        State      HasMoreData      Location       Command
--          --         --         --             --           --
9          Job9       Running    False           Server1       Get-WMIObject
PS C:\>
```

## How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. You would have needed to open an extra PowerShell session whilst you waited for a command to complete in your current session.

### Related Cmdlets

[Get-Job](#)

[Receive-Job](#)

[Wait-Job](#)

[Stop-Job](#)

[Remove-Job](#)

[Invoke-Command](#)

## #16 Get-Job

### Get-Job

#### What can I do with it?

Get background jobs from the current session as objects.

#### Examples:

Get background jobs from the current session.

#### Get-Job

Administrator: Windows PowerShell					
Id	Name	State	HasMoreData	Location	Command
1	Job1	Stopped	False	localhost	Write-Host "Start"...
9	Job9	Running	False	Server1	Get-WMIObject
PS C:\> _					

Get background jobs from the current session, which contain the Get-WmiObject cmdlet.

```
Get-Job -Command "Get-WmiObject"
```

Administrator: Windows PowerShell					
Id	Name	State	HasMoreData	Location	Command
9	Job9	Failed	False	Server1	Get-WMIObject
PS C:\> _					

Store a job in a variable and examine its methods and properties.

```
$job = Get-Job -Command "Get-WmiObject"
$job | Get-Member
```

```

Administrator: Windows PowerShell
PS C:\> $job = Get-Job -Command "Get-WMIObject"
PS C:\> $job | Get-Member

TypeName: Microsoft.PowerShell.Commands.PSWmiJob
Name          MemberType      Definition
----          --           --
StateChanged  Event          System.EventHandler`1[System.Manag...
Dispose       Method          System.Void Dispose()
Equals        Method          bool Equals(System.Object obj)
 GetHashCode   Method          int GetHashCode()
 GetType       Method          type GetType()
 StopJob      Method          System.Void StopJob()
 ToString     Method          string ToString()
 ChildJobs    Property        System.Collections.Generic.IList`1
 Command      Property        System.String Command {get;}
 Debug        Property        System.Management.Automation.PSDat...
 Error        Property        System.Management.Automation.PSDat...
 Finished     Property        System.Threading.WaitHandle Finish...
 HasMoreData   Property        System.Boolean HasMoreData {get;}
 Id           Property        System.Int32 Id {get;}
 InstanceId   Property        System.Guid InstanceId {get;}
 JobStateInfo Property        System.Management.Automation.JobSt...
 Location     Property        System.String Location {get;}
 Name         Property        System.String Name {get;set;}
 Output       Property        System.Management.Automation.PSDat...
 Progress     Property        System.Management.Automation.PSDat...
 StatusMessage Property        System.String StatusMessage {get;}
 Verbose      Property        System.Management.Automation.PSDat...
 Warning     Property        System.Management.Automation.PSDat...
 State        ScriptProperty System.Object State {get=$this.Job
PS C:\> _

```

## How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. You would have needed to open an extra PowerShell session whilst you waited for a command to complete in your current session.

### Related Cmdlets

[Receive-Job](#)

[Wait-Job](#)

[Start-Job](#)

[Stop-Job](#)

[Remove-Job](#)

[Invoke-Command](#)

## #17 Receive-Job

### Receive-Job

#### What can I do with it?

Retrieve the results of a background job which has already been run.

#### Example:

Retrieve the results for the job with ID 1 and keep them available for retrieval again. (The default is to remove them)

```
Receive-Job -Id 1 -Keep
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Get-Job is run, displaying a table with columns: Id, Name, State, HasMoreData, Location, and Command. One job, Job1, is listed with Id 1, State Completed, HasMoreData True, Location localhost, and Command Write-Host "Start". The command PS C:\> Receive-Job -Id 1 -Keep is then run, followed by PS C:\>. The output shows the command Start, indicating the job's results were received.

Id	Name	State	HasMoreData	Location	Command
1	Job1	Completed	True	localhost	Write-Host "Start"

#### How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. You would have needed to open an extra PowerShell session whilst you waited for a command to complete in your current session.

#### Related Cmdlets

[Get-Job](#)

[Wait-Job](#)

[Start-Job](#)

[Stop-Job](#)

[Remove-Job](#)

[Invoke-Command](#)

## #18 Remove-Job

### Remove-Job

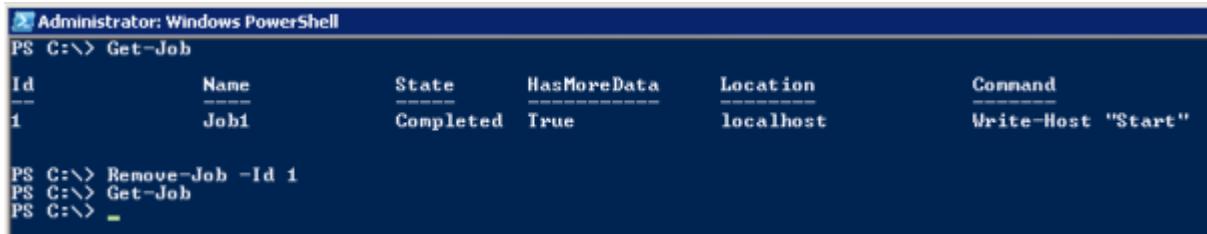
#### What can I do with it?

Remove existing background jobs from the current session.

#### Examples:

Remove the job with ID 1.

```
Remove-Job -Id 1
```

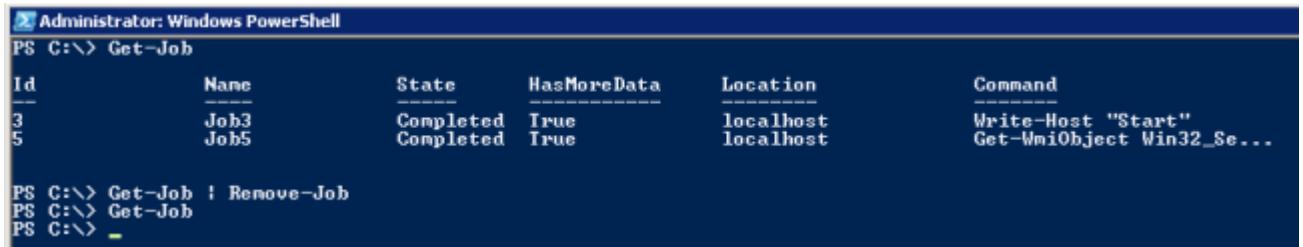


The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Job" is run, displaying a table with columns: Id, Name, State, HasMoreData, Location, and Command. One job, named "Job1", is listed with Id 1, State Completed, HasMoreData True, Location localhost, and Command Write-Host "Start". The next command, "Remove-Job -Id 1", is run, followed by another "Get-Job" command which shows no results, indicating the job has been removed.

Id	Name	State	HasMoreData	Location	Command
1	Job1	Completed	True	localhost	Write-Host "Start"

Use the [Get-Job](#) cmdlet to retrieve all jobs and pipe it through to Remove-Job to remove them all.

```
Get-Job | Remove-Job
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Job" is run, displaying two completed jobs: "Job3" and "Job5". The output of "Get-Job" is piped ("|") into the "Remove-Job" command. After the removal, another "Get-Job" command is run, showing no results.

Id	Name	State	HasMoreData	Location	Command
3	Job3	Completed	True	localhost	Write-Host "Start"
5	Job5	Completed	True	localhost	Get-WmiObject Win32_Se...

#### How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. You would have needed to open an extra PowerShell session whilst you waited for a command to complete in your current session.

#### Related Cmdlets

[Get-Job](#)

[Receive-Job](#)

[Wait-Job](#)

[Start-Job](#)

[Stop-Job](#)

## Invoke-Command

## #19 Stop-Job

### Stop-Job

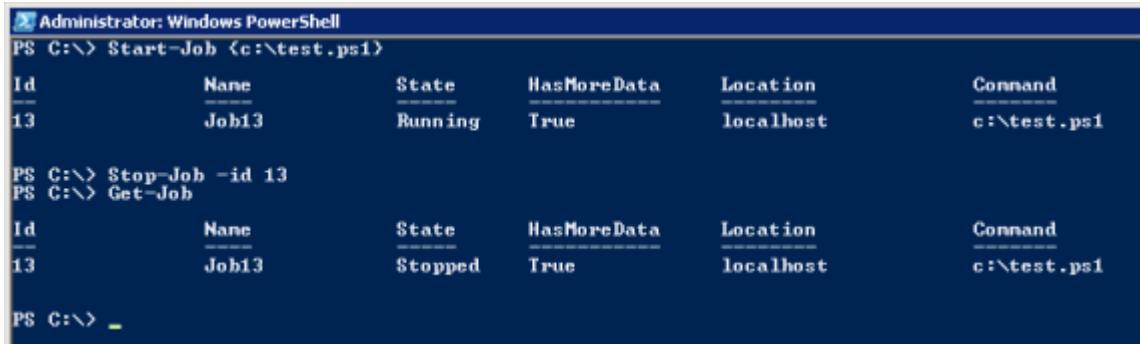
#### What can I do with it?

Stop background jobs which are running in the current session.

#### Examples:

Stop job with id 13.

```
Stop-Job -Id 13
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays the output of the following commands:

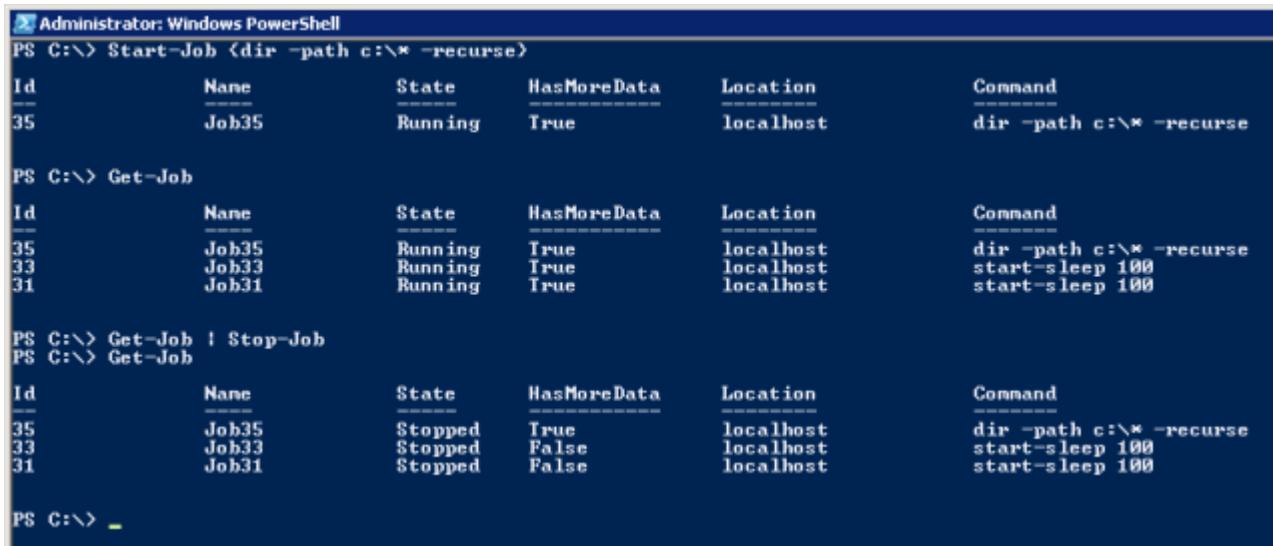
```
PS C:\> Start-Job <c:\test.ps1>
Id      Name      State      HasMoreData      Location      Command
13      Job13    Running     True          localhost      c:\test.ps1

PS C:\> Stop-Job -id 13
PS C:\> Get-Job
Id      Name      State      HasMoreData      Location      Command
13      Job13    Stopped    True          localhost      c:\test.ps1

PS C:\> _
```

Retrieve all current jobs and stop them all.

```
Get-Job | Stop-Job
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays the output of the following commands:

```
PS C:\> Start-Job <dir -path c:\* -recurse>
Id      Name      State      HasMoreData      Location      Command
35      Job35    Running     True          localhost      dir -path c:\* -recurse

PS C:\> Get-Job
Id      Name      State      HasMoreData      Location      Command
35      Job35    Running     True          localhost      dir -path c:\* -recurse
33      Job33    Running     True          localhost      start-sleep 100
31      Job31    Running     True          localhost      start-sleep 100

PS C:\> Get-Job | Stop-Job
PS C:\> Get-Job
Id      Name      State      HasMoreData      Location      Command
35      Job35    Stopped    True          localhost      dir -path c:\* -recurse
33      Job33    Stopped    False         localhost      start-sleep 100
31      Job31    Stopped    False         localhost      start-sleep 100

PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. You would have needed to open an extra PowerShell session whilst you waited for a command to complete in your current session.

#### Related Cmdlets

[Get-Job](#)

[Receive-Job](#)

[Wait-Job](#)

[Start-Job](#)

[Remove-Job](#)

[Invoke-Command](#)

## #20 Wait-Job

### [Wait-Job](#)

#### What can I do with it?

Wait for a background job to complete in the current session before returning the prompt to the user.

#### Example:

Wait for jobs 37, 39 and 41 to finish, but use the **Any** parameter to only wait for the first one. You can see when first initiated the cursor does not return to the prompt.

```
Administrator: Windows PowerShell
PS C:\> Start-Job <dir -path c:\* -recurse>
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
41          Job41      Running    True           localhost     dir -path c:\* -recurse

PS C:\> Get-Job
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
41          Job41      Running    True           localhost     dir -path c:\* -recurse
39          Job39      Running    True           localhost     start-sleep 100
37          Job37      Running    True           localhost     start-sleep 100

PS C:\> Wait-Job -id 37,39,41 -Any
-
```

As soon as one of those jobs completes the cursor returns the prompt. We then use the [Get-Job](#) cmdlet to confirm that even though Job 41 is still running we have been given the prompt back.

```

Administrator: Windows PowerShell
PS C:\> Start-Job <dir -path c:\* -recurse>
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
41          Job41      Running    True           localhost     dir -path c:\* -recurse
39          Job39      Running    True           localhost     start-sleep 100
37          Job37      Running    True           localhost     start-sleep 100

PS C:\> Get-Job
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
37          Job37      Completed  False          localhost     start-sleep 100

PS C:\> Wait-Job -id 37,39,41 -Any
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
37          Job37      Completed  False          localhost     start-sleep 100

PS C:\> Get-Job
Id          Name        State      HasMoreData      Location      Command
--          --          --          --          --          --
41          Job41      Running    True           localhost     dir -path c:\* -recurse
39          Job39      Completed  False          localhost     start-sleep 100
37          Job37      Completed  False          localhost     start-sleep 100

PS C:\> 

```

## How could I have done this in PowerShell 1.0?

The concept of jobs did not exist in PowerShell 1.0. Waiting for a command to complete before having the prompt returned to the user was standard behaviour.

## Related Cmdlets

[Get-Job](#)

[Receive-Job](#)

[Start-Job](#)

[Stop-Job](#)

[Remove-Job](#)

[Invoke-Command](#)

## #21 Select-XML

### Select-XML

#### What can I do with it?

Search for text in an XML document using an [XPath query](#).

#### Example:

Example.xml

```
<?xml version="1.0" standalone="yes" ?>
- <shop location="Birmingham" size="Large">
  - <food>
    <Name>Apple</Name>
    <type>fruit</type>
    <cost>15</cost>
  </food>
  - <food>
    <Name>Carrot</Name>
    <type>vegetable</type>
    <cost>10</cost>
  </food>
</shop>
```

From the file Example.xml search with the XPath query **/shop/food**

```
Select-XML -Path example.xml -XPath "/shop/food"
```

Administrator: Windows PowerShell		
Node	Path	Pattern
food	C:\example.xml	/shop/food
food	C:\example.xml	/shop/food

You'll notice this hasn't returned any actual data from the XML file rather details of the search carried out and two matches. This is because Select-XML returns a SelectXmlInfo Object, illustrated below by piping the same command to Get-Member.

```

Administrator: Windows PowerShell
PS C:\> Select-XML -Path example.xml -XPath "/shop/food" | Get-Member

TypeName: Microsoft.PowerShell.Commands.SelectXmlInfo

Name      MemberType  Definition
----      --          --
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
ToString  Method     string ToString()
Node     Property   System.Xml.XmlNode Node {get;set;}
Path    Property   System.String Path {get;set;}
Pattern  Property   System.String Pattern {get;set;}

PS C:\>

```

To retrieve the results, pipe the `SelectXMLInfo` object through to `Select-Object` and use the `-ExpandProperty` parameter.

```

Select-XML -Path example.xml -XPath "/shop/food"
| Select-Object -ExpandProperty Node

```

Name	type	cost
Apple	fruit	15
Carrot	vegetable	10

## How could I have done this in PowerShell 1.0?

You could have used the `Get-Content` cmdlet to read the `Example.xml` file in as text, converted it to an XML type using `[XML]` and then used the `SelectNodes` method to retrieve the data.

```

[xml]$xml = (Get-Content example.xml)
$xml.SelectNodes("/shop/food")

```

Name	type	cost
Apple	fruit	15
Carrot	vegetable	10

## Related Cmdlets

[Convert-ToXML](#)

## #22 Enable-ComputerRestore

[Enable-ComputerRestore](#)

### What can I do with it?

Enable the System Restore feature on the specified drive.

#### Example:

Enable System Restore on the local C drive.

```
Enable-ComputerRestore -Drive "C:\\"
```

### How could I have done this in PowerShell 1.0?

You could have used the **SystemRestore** [WMI class](#) and the **Enable** [method](#)

```
$SystemRestore = [wmiclass]"\\.\root\default:systemrestore"  
$SystemRestore.Enable("c:\\")
```

### Related Cmdlets

[Disable-ComputerRestore](#)

[Get-ComputerRestorePoint](#)

[Restore-Computer](#)

[Restart-Computer](#)

## #23 Disable-ComputerRestore

[Disable-ComputerRestore](#)

### What can I do with it?

Disable the System Restore feature on the specified drive.

#### Example:

Disable System Restore on the local C drive.

```
Disable-ComputerRestore -Drive "C:\\"
```

### How could I have done this in PowerShell 1.0?

You could have used the **SystemRestore** [WMI class](#) and the **Disable** [method](#)

```
$SystemRestore = [wmiclass]"\\.\root\default:systemrestore"  
$SystemRestore.Disable("c:\\")
```

### Related Cmdlets

[Enable-ComputerRestore](#)

[Get-ComputerRestorePoint](#)

[Restore-Computer](#)

[Restart-Computer](#)

## #24 Get-ComputerRestorePoint

[Get-ComputerRestorePoint](#)

### What can I do with it?

List available System Restore points on the local machine.

#### Example:

List the available System Restore points on the current machine.

[Get-ComputerRestorePoint](#)

CreationTime	Description	SequenceNumber	EventType	RestorePointType
12/11/2009 20:41:28	Windows Update	86	BEGIN_SYSTEM_C...	18
16/11/2009 20:42:24	Windows Update	87	BEGIN_SYSTEM_C...	18
17/11/2009 23:12:59	Windows Update	88	BEGIN_SYSTEM_C...	18
19/11/2009 22:20:37	Windows Update	89	BEGIN_SYSTEM_C...	18
23/11/2009 19:24:36	Windows Update	90	BEGIN_SYSTEM_C...	18
28/11/2009 09:41:18	Windows Update	91	BEGIN_SYSTEM_C...	18
30/11/2009 20:26:20	Windows Update	92	BEGIN_SYSTEM_C...	18
03/12/2009 19:39:06	Windows Update	93	BEGIN_SYSTEM_C...	18
07/12/2009 21:10:47	Windows Update	94	BEGIN_SYSTEM_C...	18
10/12/2009 07:16:25	Windows Update	95	BEGIN_SYSTEM_C...	18
11/12/2009 19:23:02	Windows Update	96	BEGIN_SYSTEM_C...	18
14/12/2009 21:15:21	Windows Update	97	BEGIN_SYSTEM_C...	18
17/12/2009 20:19:58	Windows Update	98	BEGIN_SYSTEM_C...	18

### How could I have done this in PowerShell 1.0?

You could have used the `Get-WmiObject` cmdlet with the `Root\Default` namespace and the [SystemRestore](#) Class

```
Get-WmiObject -Namespace root\default -Class SystemRestore
```

Funnily enough `Get-ComputerRestorePoint` and `Get-WmiObject -Namespace root\default -Class SystemRestore` look pretty similar when you pipe them through to `Get-Member` 😊

### Related Cmdlets

[Enable-ComputerRestore](#)

[Disable-ComputerRestore](#)

[Restore-Computer](#)

[Restart-Computer](#)

## #25 Restore-Computer

[Restore-Computer](#)

### What can I do with it?

Run a system restore on the local machine.

#### Example:

Restore the local computer to restore point 101 and then use the [Restart-Computer](#) cmdlet to reboot it

```
Restore-Computer -RestorePoint 101
Restart-Computer
```

### How could I have done this in PowerShell 1.0?

You could have used the **SystemRestore** [WMI class](#) and the **Restore** [method](#). You could then use the **Get-WmiObject** cmdlet, the **Win32\_OperatingSystem** [class](#) and the [Reboot](#) method to restart the machine.

```
$SystemRestore = [wmiclass]"\\.\root\default:systemrestore"
$SystemRestore.Restore("101")
(Get-WmiObject Win32_OperatingSystem).reboot()
```

### Related Cmdlets

[Enable-ComputerRestore](#)

[Disable-ComputerRestore](#)

[Get-ComputerRestorePoint](#)

[Restart-Computer](#)

## #26 New-WebServiceProxy

### New-WebServiceProxy

#### What can I do with it?

Make use of an available web service.

#### Examples:

The website <http://www.webservicex.net> has a number of available web services which you can use with the New-WebServiceProxy cmdlet.

#### Find the current weather for Southampton, UK.

```
$weather = New-WebServiceProxy  
-URI "http://www.webservicex.net/globalweather.asmx?wsdl"  
$weather.GetWeather('Southampton', 'United Kingdom')  
  
PS C:\> $weather = New-WebServiceProxy -uri "http://www.webservicex.net/globalweather.asmx?wsdl"  
PS C:\> $weather.GetWeather('Southampton', 'United Kingdom')  
<?xml version="1.0" encoding="utf-16"?>  
<CurrentWeather>  
  <Location>Southampton / Weather Centre, United Kingdom (EGHI) 50-54N 001-24W 0M</Location>  
  <Time>Dec 23, 2009 - 07:50 AM EST / 2009.12.23 1250 UTC</Time>  
  <Wind> Variable at 1 MPH (1 KT)</Wind>  
  <Visibility> 4 mile(s):0</Visibility>  
  <SkyConditions> mostly cloudy</SkyConditions>  
  <Temperature> 37 F (3 C)</Temperature>  
  <DewPoint> 35 F (2 C)</DewPoint>  
  <RelativeHumidity> 93%</RelativeHumidity>  
  <Pressure> 29.21 in. Hg (998 hPa)</Pressure>  
  <Status>Success</Status>  
</CurrentWeather>  
PS C:\>
```

Note: to find what cities were available within the UK to query I used the **GetCitiesByCountry** method.

```
$weather.GetCitiesByCountry('United Kingdom')
```

#### Find the location for the UK postcode SW1A.

```
$postcode = New-WebServiceProxy  
-URI "http://www.webservicex.net/uklocation.asmx?wsdl"  
$postcode.GetUKLocationByPostcode('SW1A')  
  
PS C:\> $postcode = New-WebServiceProxy -uri "http://www.webservicex.net/uklocation.asmx?wsdl"  
PS C:\> $postcode.GetUKLocationByPostcode('SW1A')  
<NewDataSet>  
  <Table>  
    <Town>Westminster Abbey</Town>  
    <County>Greater London</County>  
    <PostCode>SW1A</PostCode>  
  </Table>  
</NewDataSet>  
PS C:\>
```

#### Return the words from Bible verse John, Chapter 3, 16.

```
$bibleverse = New-WebServiceProxy  
-URI "http://www.webservicex.net/BibleWebservice.asmx?wsdl"  
$bibleverse.GetBibleWordsByChapterAndVerse('John', '3', '16')
```

```

PS C:\> $bibleverse = New-WebServiceProxy -uri "http://www.webservicex.net/BibleWebservice.asmx?wsdl"
PS C:\> $bibleverse.GetBibleWordsByChapterAndVerse('John','3','16')
<NewDataSet>
  <Table>
    <Book>43</Book>
    <BookTitle>John</BookTitle>
    <Chapter>3</Chapter>
    <Verse>16</Verse>
    <BibleWords>For God so loved the world, that he gave his only begotten Son, that whosoever believeth in him should
not perish, but have everlasting life.</BibleWords>
  </Table>
</NewDataSet>
PS C:\>

```

There are many more web services available from <http://www.webservicex.net> which are fun to try out. You also may find other available web services on the Internet or within your own organisation.

### **How could I have done this in PowerShell 1.0?**

Lee Holmes from the PowerShell team has put together a [Connect-WebService script](#) you can use when working with PowerShell 1.0 to make use a web service. By using this script you can follow a similar process to the earlier examples.

```

$weather = .\Connect-WebService.ps1
"http://www.webservicex.net/globalweather.asmx?wsdl"
$weather.GetWeather('Southampton', 'United Kingdom')

```

I reckon that's about 100 lines of code in a script down to one nice easy cmdlet when using the New-WebServiceProxy cmdlet. Nice!

## #27 Test-ComputerSecureChannel

[Test-ComputerSecureChannel](#)

### What can I do with it?

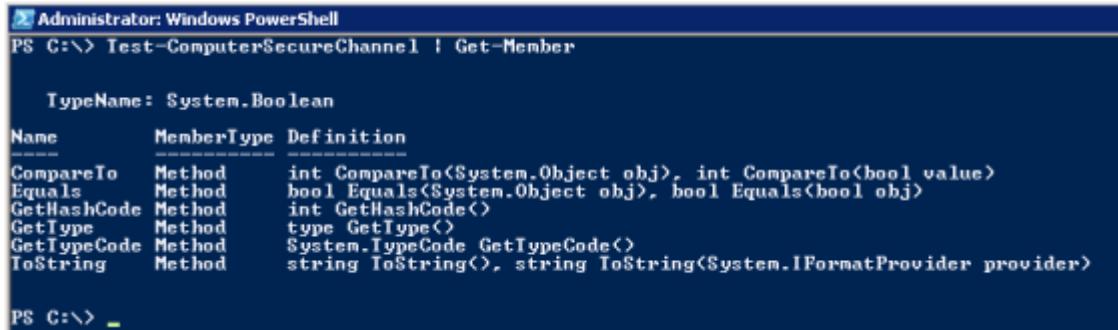
Test the secure channel between the local computer and the domain and optionally fix if necessary.

#### Example:

Test the secure channel on the current computer

[Test-ComputerSecureChannel](#)

Note: this will return a boolean value of True or False as seen below; if you wish for more detailed information use the -Verbose parameter.



```
Administrator: Windows PowerShell
PS C:\> Test-ComputerSecureChannel | Get-Member

 TypeName: System.Boolean
Name      MemberType Definition
----      --          --
CompareTo Method   int CompareTo(System.Object obj), int CompareTo(bool value)
Equals    Method   bool Equals(System.Object obj), bool Equals(bool obj)
GetHashCode Method   int GetHashCode()
GetType   Method   type GetType()
GetTypeCode Method   System.TypeCode GetTypeCode()
ToString  Method   string ToString(), string ToString(System.IFormatProvider provider)

PS C:\> _
```

If the result is False then you can attempt to fix the problem by using the -Repair parameter.

[Test-ComputerSecureChannel -Repair](#)

### How could I have done this in PowerShell 1.0?

The best way to do it would have been using the **NetDom** command line tool.

#### Test:

```
netdom verify Server01 /domain:test.local
```

#### Repair:

```
netdom reset Server01 /domain:test.local
```

### Related Cmdlets

[Checkpoint-Computer](#)

[Restart-Computer](#)

[Stop-Computer](#)

[Reset-ComputerMachinePassword](#)

## #28 Export-Counter

### [Export-Counter](#)

#### What can I do with it?

Take performance objects generated from the [Get-Counter](#) or [Import-Counter](#) cmdlets and export them as log files. **Note:** this cmdlet requires Windows 7 or Windows Server 2008 R2 or later.

#### Examples:

Retrieve some memory performance data from the local machine and export it to the standard Performance Monitor output file BLG.

```
Get-Counter '\Memory\Pool Paged Bytes' -MaxSamples 10 |  
Export-Counter -Path C:\Memory.blg
```

You can also output directly to two other format types, CSV and TSV.

```
Get-Counter '\Memory\Pool Paged Bytes' -MaxSamples 10 |  
Export-Counter -Path C:\Memory.csv -FileFormat CSV
```

#### How could I have done this in PowerShell 1.0?

In the [Get-Counter](#) post I showed an example using .NET to retrieve performance data, but it would only return one result at a time, so not a lot of point to extract to a log file.

```
$data = New-Object System.Diagnostics.PerformanceCounter  
$data.CategoryName = "Memory"  
$data.CounterName = "Pool Paged Bytes"  
$data.nextvalue()
```

You could run the final command multiple times and output to a text file, but still not a particularly nice solution.

```
for ($i=1; $i -le 10; $i++) {$data.nextvalue() | Out-File test.txt -Append}
```

Alternatively from the Performance Monitor GUI you could create a Data Collector Set to save performance data into a BLG file. You could then use the [Relog.exe](#) tool to convert the BLG file into CSV or TSV.

#### Related Cmdlets

##### [Get-Counter](#)

##### [Import-Counter](#)

## #29 Import-Counter

### [Import-Counter](#)

#### **What can I do with it?**

Create objects by importing performance data in BLG, CSV or TSV files.

#### **Example:**

Import as objects data in a BLG file previously exported from [Export-Counter](#) or the Performance Monitor GUI.

```
$performanceData = Import-Counter -Path Memory.blg
```

#### **How could I have done this in PowerShell 1.0?**

To manage performance data contained in a BLG file you could have used the Performance Monitor GUI to import it and view the contents.

#### **Related Cmdlets**

##### [Get-Counter](#)

##### [Export-Counter](#)

## #30 Enable-PSRemoting

### Enable-PSRemoting

#### What can I do with it?

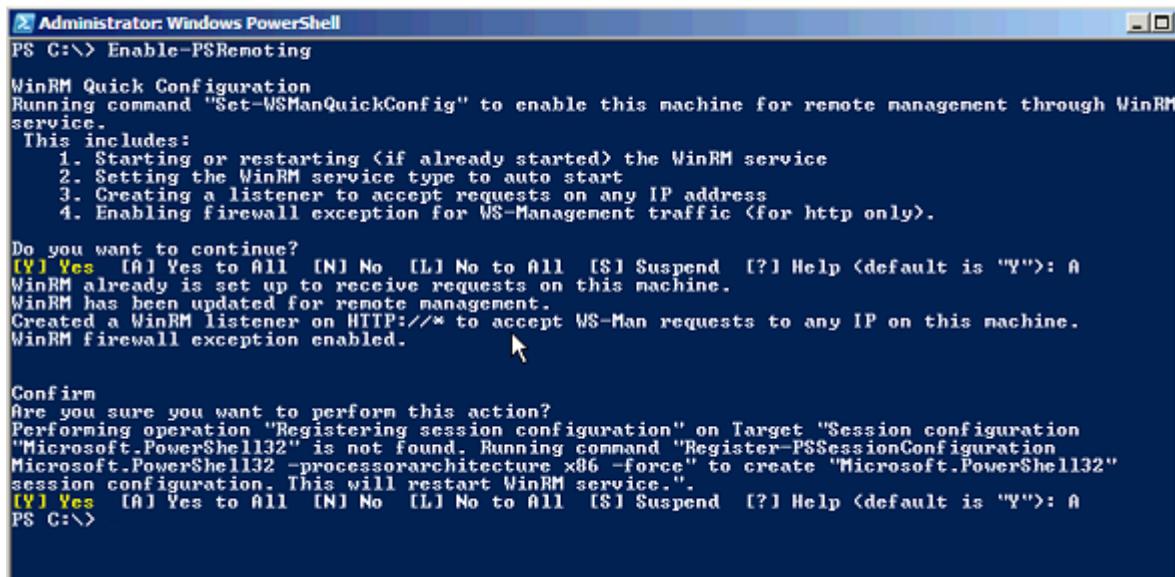
Configure a computer to be enabled for PowerShell remoting. **Tip:** Make sure you run this cmdlet from an elevated process.

#### Example:

Configure the computer Test01 to be enabled for PowerShell remoting.

### Enable-PSRemoting

This will produce output similar to the below; note the command was run on a Windows Server 2008 64-bit system



```
Administrator: Windows PowerShell
PS C:\> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable this machine for remote management through WinRM service.
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service type to auto start
  3. Creating a listener to accept requests on any IP address
  4. Enabling Firewall exception 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">: A
WinRM already is set up to receive requests on this machine.
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 operation "Registering session configuration" on Target "Session configuration "Microsoft.PowerShell132" is not found. Running command "Register-PSSessionConfiguration Microsoft.PowerShell132 -processorarchitecture x86 -force" to create "Microsoft.PowerShell132" session configuration. This will restart WinRM service.".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help <default is "Y">: A
PS C:\>
```

You will notice from the output that it runs two other PowerShell 2.0 cmdlets, Set-WSManQuickConfig and Register-PSSessionConfiguration. The below (taken from PowerShell help) gives a great summary of what each will do in this instance.

- Runs the Set-WSManQuickConfig cmdlet, which performs the following tasks:
  - Starts the WinRM service.
  - Sets the startup type on the WinRM service to Automatic.
  - Creates a listener to accept requests on any IP address.
  - Enables a firewall exception for WS-Management communications.
  
- Enables all registered Windows PowerShell session configurations to receive instructions from a remote computer.
  - Registers the "Microsoft.PowerShell" session configuration, if it is not already registered.
  - Registers the "Microsoft.PowerShell32" session configuration on 64-bit computers, if it

is not already registered.

----- Removes the "Deny Everyone" setting from the security descriptor for all the registered session configurations.

----- Restarts the WinRM service to make the preceding changes effective.

## **How could I have done this in PowerShell 1.0?**

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

### **Related Cmdlets**

[Disable-PSRemoting](#)

[Get-PSSessionConfiguration](#)

[Enable-PSSessionConfiguration](#)

[Disable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

## #31 Enter-PSSession

### Enter-PSSession

#### What can I do with it?

Open an interactive PowerShell session with a computer which has been enabled for PowerShell remoting.

#### Example:

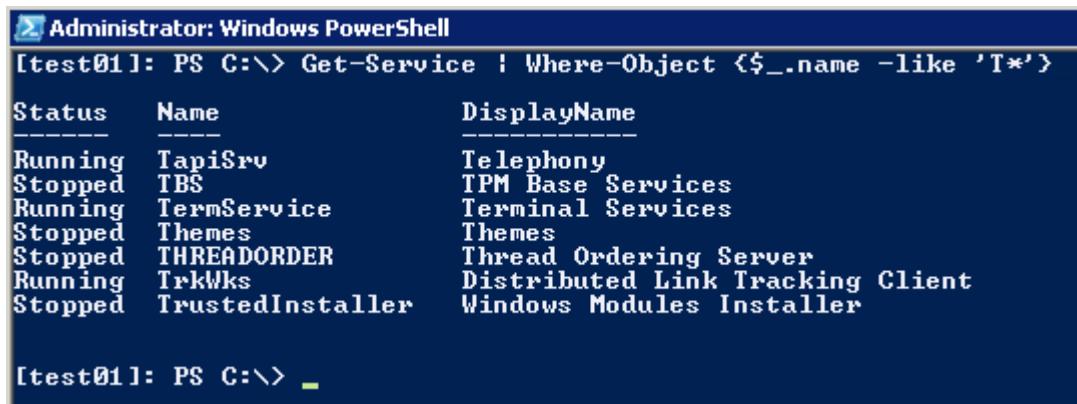
Open a session with the server Test01 and see which services begin with the letter T.

```
Enter-PSSession -ComputerName Test01
Get-Service | Where-Object {$_.name -like 'T*'}
```

You will notice that the prompt has changed to

[test01]: PS C:>

which helpfully shows you which server you are running the remote session on.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was "Get-Service | Where-Object {\$\_.name -like 'T\*'}", which lists all services starting with the letter 'T'. The output is as follows:

Status	Name	DisplayName
Running	TapiSrv	Telephony
Stopped	TBS	TPM Base Services
Running	TermService	Terminal Services
Stopped	Themes	Themes
Stopped	THREADORDER	Thread Ordering Server
Running	TrkWks	Distributed Link Tracking Client
Stopped	TrustedInstaller	Windows Modules Installer

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[New-PSSession](#)

[Get-PSSession](#)

[Exit-PSSession](#)

[Remove-PSSession](#)

[Invoke-Command](#)

## #32 Exit-PSSession

### [Exit-PSSession](#)

#### What can I do with it?

Exit an interactive PowerShell session that has been opened on a computer which has been enabled for PowerShell remoting.

#### Example:

Leave an interactive PowerShell session with a computer which has been enabled for PowerShell remoting.

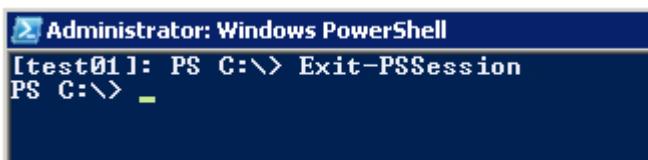
#### [Exit-PSSession](#)

You will notice that the prompt has changed back from

[test01]: PS C:\>

to simply

PS C:\>



```
[test01]: PS C:\> Exit-PSSession
PS C:\> _
```

**Tip:** When in an interactive remote session you can also just type **Exit** to finish the session.

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[New-PSSession](#)

[Get-PSSession](#)

[Enter-PSSession](#)

[Remove-PSSession](#)

[Invoke-Command](#)

## #33 New-PSSession

### New-PSSession

#### What can I do with it?

Establish a persistent connection to a computer that has been enabled for PowerShell remoting.

#### Examples:

Establish a persistent remote PowerShell connection to Test01 and store it in the variable **\$session**. Then use the [Enter-PSSession](#) cmdlet with the **Session** parameter to use that session.

```
$session = New-PSSession -ComputerName Test01  
Enter-PSSession -Session $session
```

You can also open multiple sessions via different methods:

Open sessions to Test01, Test02 and Test 03.

```
$session1, $session2, $session3 =  
New-PSSession -ComputerName Test01,Test02,Test03
```

Or if you have the servers stored in a csv file.

```
$sessions = New-PSSession -ComputerName (Get-Content servers.csv)
```

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[Get-PSSession](#)

[Enter-PSSession](#)

[Exit-PSSession](#)

[Remove-PSSession](#)

[Invoke-Command](#)

## #34 Invoke-Command

### Invoke-Command

#### What can I do with it?

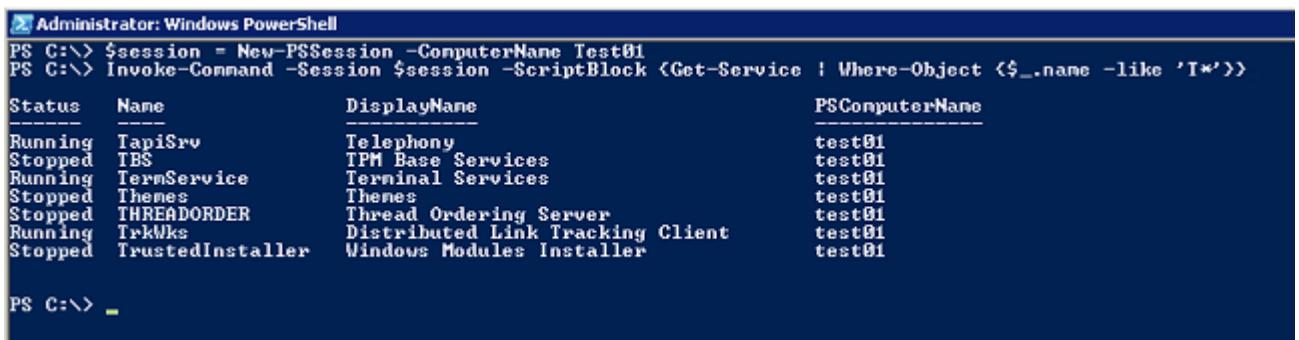
Run commands on local or remote computers and return the results.

#### Examples:

Establish a persistent remote PowerShell connection to Test01 using [New-PSSession](#) and store it in the variable **\$session**. Then return the results for which services begin with T.

```
$session = New-PSSession -ComputerName Test01
Invoke-Command -Session $session -ScriptBlock {Get-Service | Where-Object {$_.name -like 'T*'}}
```

You can see that the results contain a property **PSComputerName** that shows which server the results came from.



Status	Name	DisplayName	PSComputerName
Running	TapiSrv	Telephony	test01
Stopped	TBS	TPM Base Services	test01
Running	TermService	Terminal Services	test01
Stopped	Themes	Themes	test01
Stopped	THREADORDER	Thread Ordering Server	test01
Running	TrkWks	Distributed Link Tracking Client	test01
Stopped	TrustedInstaller	Windows Modules Installer	test01

You don't need to stretch your imagination too far to see how this could quickly become extremely powerful. Imagine instead that you used [New-PSSession](#) to make sessions to 100 servers stored in a csv file and run the same command to all of those servers. The change in the code would be very small.

```
$sessions = New-PSSession -ComputerName (Get-Content servers.csv)
Invoke-Command -Session $sessions -ScriptBlock
{Get-Service | Where-Object {$_.name -like 'T*'}}
```

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[New-PSSession](#)

[Get-PSSession](#)

[Enter-PSSession](#)

[Exit-PSSession](#)

[Remove-PSSession](#)

## #35 New-PSSessionOption

### New-PSSessionOption

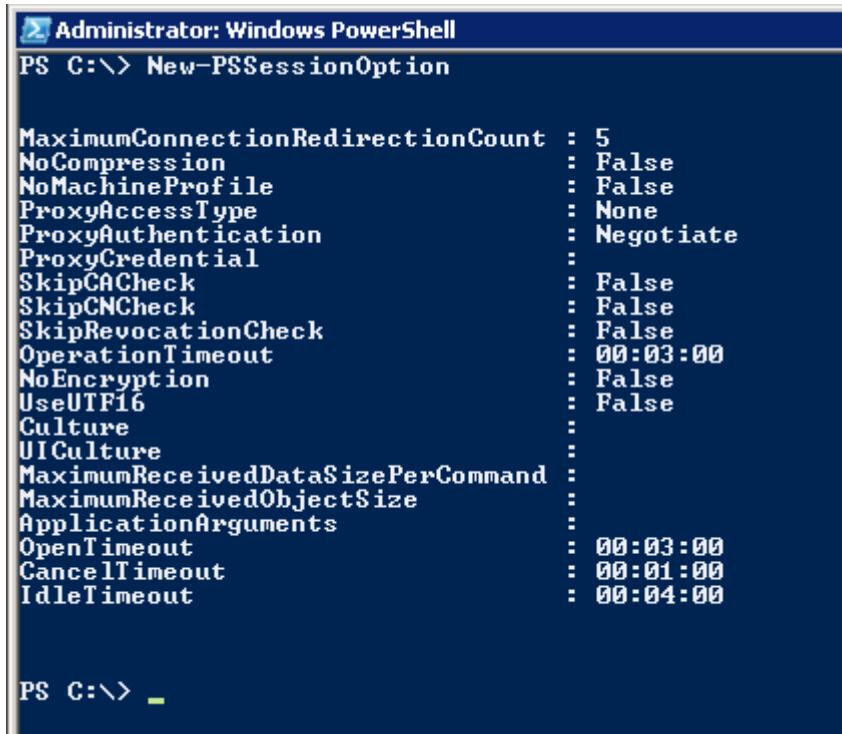
#### What can I do with it?

Create a new object with advanced session settings to be used when opening PowerShell remote sessions.

#### Examples:

Show the possible options which can be set with New-PSSessionOption

```
New-PSSessionOption
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "New-PSSessionOption" is run at the prompt. The output displays a large number of session configuration parameters and their default values. Some key entries include:

Parameter	Value
MaximumConnectionRedirectionCount	5
NoCompression	False
NoMachineProfile	False
ProxyAccessType	None
ProxyAuthentication	Negotiate
ProxyCredential	
SkipCACheck	False
SkipCNCheck	False
SkipRevocationCheck	False
OperationTimeout	00:03:00
NoEncryption	False
UseUTF16	False
Culture	
UICulture	
MaximumReceivedDataSizePerCommand	
MaximumReceivedObjectSize	
ApplicationArguments	
OpenTimeout	00:03:00
CancelTimeout	00:01:00
IdleTimeout	00:04:00

At the bottom of the window, the prompt "PS C:\> \_" is visible.

Set some advanced session options via the **\$sessionoptions** variable and use them to make a remote PowerShell connection.

```
$sessionoptions = New-PSSessionOption  
-IdleTimeout 600000 -NoCompression -NoMachineProfile  
New-PSSession -ComputerName Test01 -SessionOption $sessionoptions
```

Notice the difference from the default in the options which have been set.

```

Administrator: Windows PowerShell
PS C:\> $sessionoptions = New-PSSessionOption -IdleTimeout 600000 -NoCompression -NoMachineProfile
PS C:\> $sessionoptions

MaximumConnectionRedirectionCount : 5
NoCompression : True
NoMachineProfile : True
ProxyAccessType : None
ProxyAuthentication : Negotiate
ProxyCredential :
SkipCACheck : False
SkipCNCheck : False
SkipRevocationCheck : False
OperationTimeout : 00:03:00
NoEncryption : False
UseUTF16 : False
Culture :
UICulture :
MaximumReceivedDataSizePerCommand :
MaximumReceivedObjectSize :
ApplicationArguments :
OpenTimeout : 00:03:00
CancelTimeout : 00:01:00
IdleTimeout : 00:10:00

PS C:\> New-PSSession -ComputerName Test01 -SessionOption $sessionoptions
Id Name ComputerName State ConfigurationName Availability
3 Session3 test01 Opened Microsoft.PowerShell Available

PS C:\>

```

## How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

## Related Cmdlets

[New-PSSession](#)

[Enter-PSSession](#)

[Invoke-Command](#)

## #36 Get-PSSession

### Get-PSSession

#### What can I do with it?

Retrieve remote PowerShell sessions created in the current session.

#### Examples:

Get all current sessions

```
Get-PSSession
```

Administrator: Windows PowerShell				
PS C:\> Get-PSSession				
Id	Name	ComputerName	State	ConfigurationName
1	Session1	test01	Opened	Microsoft.PowerShell
2	Session2	test01	Opened	Microsoft.PowerShell
3	Session3	test01	Opened	Microsoft.PowerShell

```
PS C:\> _
```

Get session 3.

```
Get-PSSession -Id 3
```

Administrator: Windows PowerShell				
PS C:\> Get-PSSession -Id 3				
Id	Name	ComputerName	State	ConfigurationName
3	Session3	test01	Opened	Microsoft.PowerShell

```
PS C:\> _
```

Get all sessions open with Test01. (Not well illustrated in this screenshot since there is only one server with sessions open, but you get the idea)

```
Get-PSSession -ComputerName Test01
```

Administrator: Windows PowerShell				
PS C:\> Get-PSSession -ComputerName Test01				
Id	Name	ComputerName	State	ConfigurationName
1	Session1	test01	Opened	Microsoft.PowerShell
2	Session2	test01	Opened	Microsoft.PowerShell
3	Session3	test01	Opened	Microsoft.PowerShell

```
PS C:\> _
```

## **How could I have done this in PowerShell 1.0?**

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

### **Related Cmdlets**

[New-PSSession](#)

[Enter-PSSession](#)

[Exit-PSSession](#)

[Remove-PSSession](#)

[Invoke-Command](#)

## #37 Remove-PSSession

### Remove-PSSession

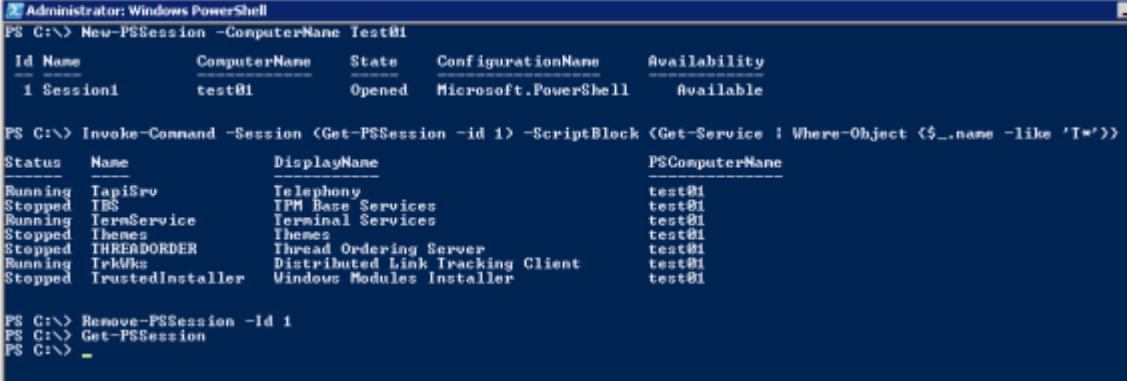
#### What can I do with it?

Close a remote PowerShell session which is open in the current session.

#### Examples:

Establish a persistent remote PowerShell connection to Test01 using [New-PSSession](#) , return the results for which services begin with T, then remove that session. Finally confirm the session has been removed by running [Get-PSSession](#) and seeing no results.

```
New-PSSession -ComputerName Test01
Invoke-Command -Session (Get-PSSession -Id 1)
-ScriptBlock {Get-Service | Where-Object {$_.name -like 'T*'}}
```



The screenshot shows a Windows PowerShell window with the following command history:

```
PS C:\> New-PSSession -ComputerName Test01
PS C:\> Invoke-Command -Session (Get-PSSession -Id 1) -ScriptBlock {Get-Service | Where-Object {$_.name -like 'T*'}}
PS C:\> Remove-PSSession -Id 1
PS C:\> Get-PSSession
PS C:\>
```

The output of the `Invoke-Command` command is a table showing services on the remote machine:

Status	Name	DisplayName	PSComputerName
Running	TapiSrv	Telephony	test01
Stopped	IIS	IPM Base Services	test01
Running	TermService	Terminal Services	test01
Stopped	Themes	Themes	test01
Stopped	THREADORDER	Thread Ordering Server	test01
Running	IrkWks	Distributed Link Tracking Client	test01
Stopped	TrustedInstaller	Windows Modules Installer	test01

An interesting thing to note is that if you store the session in a variable and then remove the session you will see that the **State** changes to **Closed**.

Establish a persistent remote PowerShell connection to Test01 using [New-PSSession](#) and store that in the variable `$session1`, confirm the session is running via [Get-PSSession](#), then remove that session. Now examine `$session1` and note the **State** as **Closed**.

```
$session1 = New-PSSession -ComputerName Test01
Get-PSSession
Remove-PSSession -Id 1
$session1
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The session starts with creating a new PSSession:

```
PS C:\> $session1 = New-PSSession -ComputerName Test01
```

Then it lists the sessions:

```
PS C:\> Get-PSSession
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	test01	Opened	Microsoft.PowerShell	Available

Finally, it removes the session:

```
PS C:\> Remove-PSSession -Id 1
```

```
PS C:\> $session1
```

Id	Name	ComputerName	State	ConfigurationName	Availability
1	Session1	test01	Closed	Microsoft.PowerShell	None

## How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

## Related Cmdlets

[New-PSSession](#)

[Get-PSSession](#)

[Enter-PSSession](#)

[Exit-PSSession](#)

[Invoke-Command](#)

## #38 Get-PSSessionConfiguration

### [Get-PSSessionConfiguration](#)

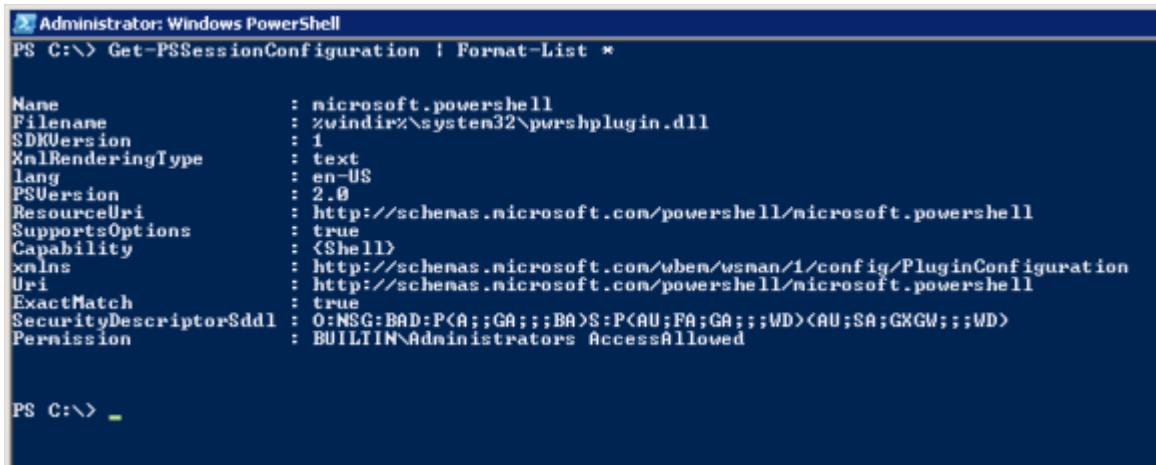
#### What can I do with it?

Session configurations determine the settings used by remote PowerShell sessions to that computer. This cmdlet displays the settings for the current configuration(s) used on the local computer.

#### Example:

Retrieve the settings used by remote PowerShell sessions on the local computer and display the properties available.

```
Get-PSSessionConfiguration | Format-List *
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-PSSessionConfiguration | Format-List \*" is run at the prompt. The output lists various session configuration properties with their values:

Property	Value
Name	: microsoft.powershell
Filename	: %windir%\system32\powershell.dll
SDKVersion	: 1
XmlRenderingType	: text
Lang	: en-US
PSVersion	: 2.0
ResourceUri	: http://schemas.microsoft.com/powershell/microsoft.powershell
SupportsOptions	: true
Capability	: <Shell>
xmlns	: http://schemas.microsoft.com/wbem/wsman/1/config/PluginConfiguration
Uri	: http://schemas.microsoft.com/powershell/microsoft.powershell
ExactMatch	: true
SecurityDescriptorSddl	: O:MSG:BAD:P(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)<AU;SA;GXGW;;;WD>
Permission	: BUILTIN\Administrators AccessAllowed

PS C:\> \_

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[Disable-PSSessionConfiguration](#)

[Enable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #39 Register-PSSessionConfiguration

### Register-PSSessionConfiguration

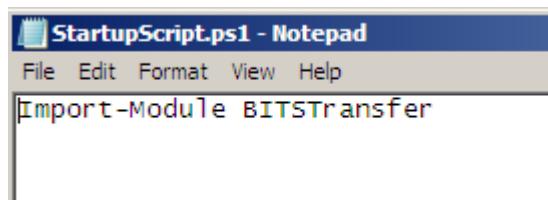
#### What can I do with it?

Session configurations determine the settings used by remote PowerShell sessions to that computer. This cmdlet enables the creation of customised settings for particular session requirements.

#### Example:

Create a new PSSession Configuration called **BITSTransfer** using the startup script C:\Scripts\StartupScript.ps1. Use StartupScript.ps1 to import the PowerShell 2.0 BITS Transfer module so that those cmdlets are available to the user of the remote session. Use [Get-PSSessionConfiguration](#) to confirm the creation.

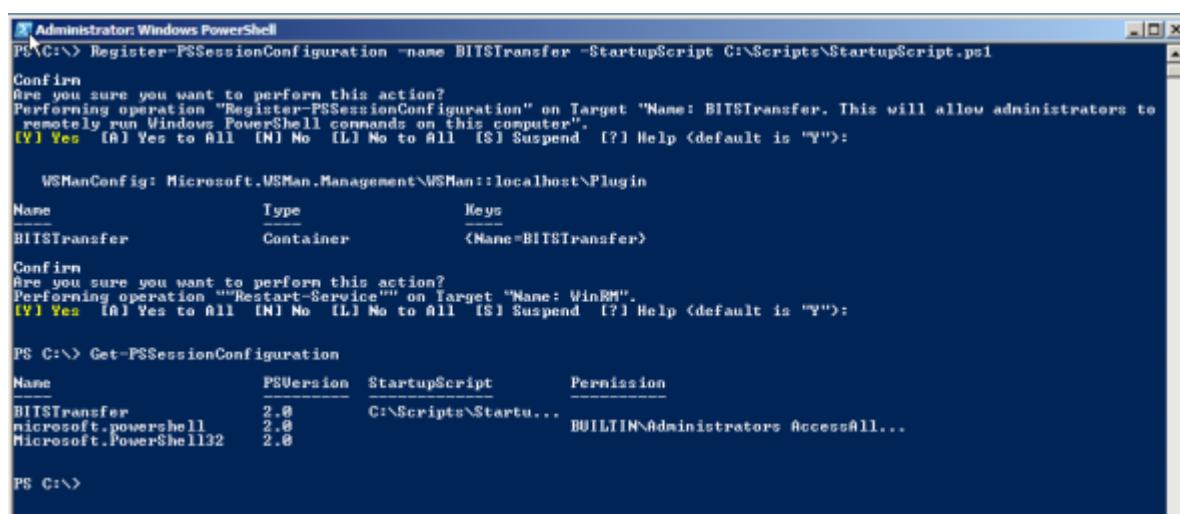
StartupScript.ps1 contains the command to import the BITSTransfer module - you could easily add other code in here to further customise the session.



```
StartupScript.ps1 - Notepad
File Edit Format View Help
Import-Module BITSTransfer
```

```
Register-PSSessionConfiguration -Name BITSTransfer
-StartupScript C:\Scripts\StartupScript.ps1
```

You will see that you are prompted for both confirmation and whether to restart the WinRM service.



```
Administrator: Windows PowerShell
PS C:\> Register-PSSessionConfiguration -name BITSTransfer -StartupScript C:\Scripts\StartupScript.ps1
Confirm
Are you sure you want to perform this action?
Performing operation "Register-PSSessionConfiguration" on Target "Name: BITSTransfer. This will allow administrators to
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">>:

WSMANConfig: Microsoft.WSMAN.Management\WSMan::localhost\Plugin
Name          Type          Keys
BITSTransfer   Container    {Name=BITSTransfer}

Confirm
Are you sure you want to perform this action?
Performing operation "Restart-Service" on Target "Name: WinRM".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help <default is "Y">>:

PS C:\> Get-PSSessionConfiguration
Name          PSVersion  StartupScript      Permission
BITSTransfer   2.0        C:\Scripts\Startu...
Microsoft.PowerShell 2.0        ...
Microsoft.PowerShell32 2.0        BUILTIN\Administrators AccessAll...
```

To use this particular PSSession Configuration use the [New-PSSession](#) cmdlet with the **ConfigurationName** parameter and specify the name of the configuration **BITSTransfer**.

Then connect to the session with the [Enter-PSSession](#) cmdlet and confirm you have the BITS Transfer module by running [Get-Module](#).

```
New-PSSession -ConfigurationName BITSTransfer  
-ComputerName Test01  
Enter-PSSession -Id 1  
Get-Module
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "New-PSSession -ConfigurationName BITSTransfer -ComputerName Test01" is run, creating a session named "Session1" on "test01" with state "Opened" and configuration "BITSTransfer". The availability is "Available". Then, "Enter-PSSession -Id 1" is run to enter the session. Finally, "Get-Module" is run, showing the "BitsTransfer" module is available with exported commands like Start-BitsTransfer, Remove-BitsTransfer, Resume-BitsTransfer, Get-Bits...

## How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

## Related Cmdlets

[Disable-PSSessionConfiguration](#)

[Enable-PSSessionConfiguration](#)

[Get-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #40 Set-PSSessionConfiguration

### [Set-PSSessionConfiguration](#)

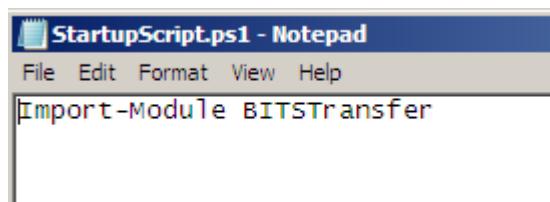
#### What can I do with it?

Change the properties of a session configuration which has been registered with [Register-PSSessionConfiguration](#).

#### Example:

Create a new PSSession Configuration called **BITSTransfer** using the startup script C:\Scripts\StartupScript.ps1. Use StartupScript.ps1 to import the PowerShell 2.0 BITS Transfer module so that those cmdlets are available to the user of the remote session. Use [Get-PSSessionConfiguration](#) to confirm the creation.

StartupScript.ps1 contains the command to import the BITSTransfer module – you could easily add other code in here to further customise the session.



Now change the properties of that session with Set-PSSessionConfiguration. Clear the startup script settings and set the **MaximumReceivedObjectSizeMB** to **50 MB**. Confirm the changes with [Get-PSSessionConfiguration](#).

```
Register-PSSessionConfiguration -Name BITSTransfer
-StartupScript C:\Scripts\StartupScript.ps1
Get-PSSessionConfiguration
Set-PSSessionConfiguration -Name BITSTransfer
-StartupScript $null -MaximumReceivedObjectSizeMB 50
Get-PSSessionConfiguration
```

You will notice that you are prompted to both confirm the change and to restart the WinRM service.

```

Administrator: Windows PowerShell
PS C:\> Set-PSSessionConfiguration -name BITSTransfer -StartupScript $null -MaximumReceivedObjectSizeMB 50
Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name: BITSTransfer".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help <default is "Y">:

    USManConfig: Microsoft.USMan.Management\USMan::localhost\Plugin\BITSTransfer\InitializationParameters
ParanName          ParanValue
psmaxinunreceived... 50

Confirm
Are you sure you want to perform this action?
Performing operation "'Restart-Service'" on Target "Name: WinRM".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help <default is "Y">:

PS C:\> Get-PSSessionConfiguration
Name          PSVersion  StartupScript      Permission
BITSTransfer   2.0
microsoft.powershell  2.0
Microsoft.PowerShell32  2.0
                           BUILTIN\Administrators AccessAll...

PS C:\>

```

## How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

## Related Cmdlets

[Disable-PSSessionConfiguration](#)

[Enable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Get-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #41 Disable-PSSessionConfiguration

### Disable-PSSessionConfiguration

#### What can I do with it?

Deny access to a session configuration.

#### Example:

Examine the permissions of the previously created PSSessionConfiguration named BITSTransfer. Deny access to this session using Disable-PSSessionConfiguration. Use the **Force** parameter to suppress prompts. Check what the permissions on the configuration have been changed to.

```
Get-PSSessionConfiguration -Name BITSTransfer  
| Format-Table -Property Name,Permission -Auto  
Disable-PSSessionConfiguration -Name BITSTransfer -Force  
Get-PSSessionConfiguration -Name BITSTransfer  
| Format-Table -Property Name,Permission -Auto
```

You will see that you are warned that disabling the PSSessionConfiguration will not undo every change made by [Enable-PSRemoting](#). The effect of running Disable-PSSessionConfiguration is to set the permission **Everyone AccessDenied**, except for **BUILTIN\Administrators Access Allowed**.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `Get-PSSessionConfiguration -Name BITSTransfer | Format-Table -Property Name,Permission -Auto` is run, showing the initial permissions:

Name	Permission
BITSTransfer	BUILTIN\Administrators AccessAllowed

The command `Disable-PSSessionConfiguration -Name BITSTransfer -Force` is then run, displaying a warning message:

WARNING: Disabling the session configurations does not undo all the changes made by the Enable-PSRemoting or Enable-PSSessionConfiguration cmdlet. You might have to manually undo the changes by following these steps:  
1. Stop and disable the WinRM service.  
2. Delete the listener that accepts requests on any IP address.  
3. Disable the firewall exceptions for WS-Management communications.  
4. Restore the value of the LocalAccountTokenFilterPolicy to 0, which restricts remote access to members of the Administrators group on the computer.

Finally, the command `Get-PSSessionConfiguration -Name BITSTransfer | Format-Table -Property Name,Permission -Auto` is run again, showing the updated permissions:

Name	Permission
BITSTransfer	Everyone AccessDenied, BUILTIN\Administrators AccessAllowed

Subsequently attempting to access that configuration from a remote session results in the following **Access Denied** error.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `New-PSSession -ConfigurationName BITSTransfer -ComputerName Test01` is run, resulting in an error message:

(test01) Connecting to remote server failed with the following error message : Access is denied. For more information, see the about\_Remote\_Troubleshooting Help topic.  
\* CategoryInfo : OpenError: (System.Management.RemoteRunspace:RemoteRunspace) [], PSRemotingTransportException  
\* FullyQualifiedErrorId : PSSessionOpenFailed

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

## **Related Cmdlets**

[Enable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #42 Enable-PSSessionConfiguration

### [Enable-PSSessionConfiguration](#)

#### What can I do with it?

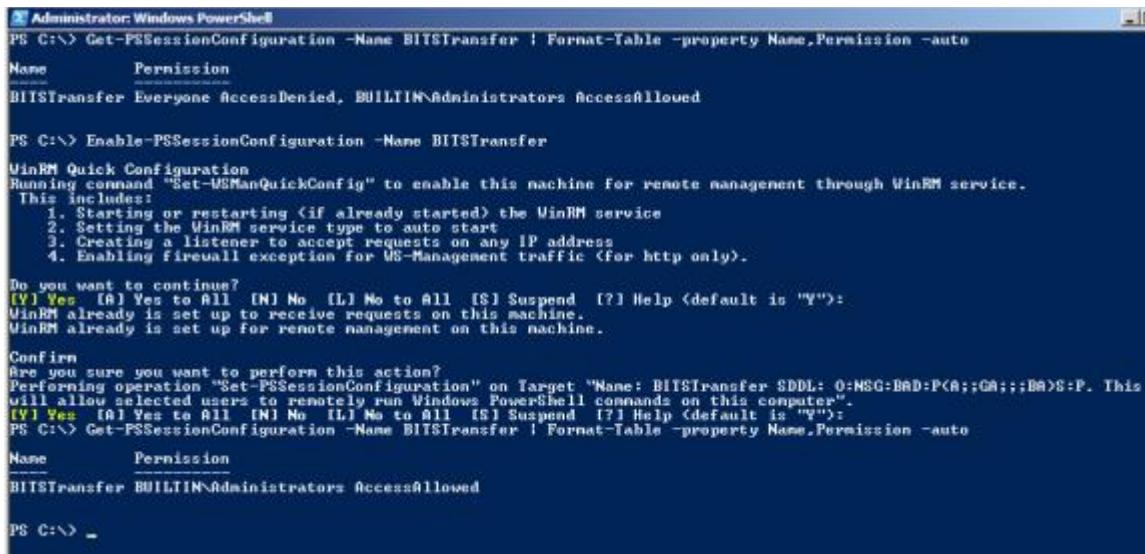
Re-enable access to a session configuration which has previously been disabled with [Disable-PSSessionConfiguration](#).

#### Example:

View the permissions of the currently disabled **BITSTransfer** PSSessionConfiguration, re-enable it, and then view the permissions again.

```
Get-PSSessionConfiguration -Name BITSTransfer  
| Format-Table -Property Name,Permission -Auto  
Enable-PSSessionConfiguration -Name BITSTransfer  
Get-PSSessionConfiguration -Name BITSTransfer  
| Format-Table -Property Name,Permission -Auto
```

You will notice that the **Everyone AccessDenied** permission is removed as part of the process, which also includes two confirmation prompts.



```
Administrator: Windows PowerShell  
PS C:\> Get-PSSessionConfiguration -Name BITSTransfer | Format-Table -property Name,Permission -auto  
Name          Permission  
BITSTransfer  Everyone AccessDenied, BUILTIN\Administrators AccessAllowed  
  
PS C:\> Enable-PSSessionConfiguration -Name BITSTransfer  
WinRM Quick Configuration  
Running command "Set-WSManQuickConfig" to enable this machine for remote management through WinRM service.  
This includes:  
1. Starting or restarting (if already started) the WinRM service  
2. Setting the WinRM service type to auto start  
3. Creating a listener to accept requests on any IP address  
4. Enabling Firewall exception 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">:  
WinRM already is set up to receive requests on this machine.  
Confirm  
Are you sure you want to perform this action?  
Performing operation "Set-PSSessionConfiguration" on Target "Name: BITSTransfer SDDL: O:MSG-BAD:PKA;;BA>S:P. This  
will allow selected users to 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">:  
PS C:\> Get-PSSessionConfiguration -Name BITSTransfer | Format-Table -property Name,Permission -auto  
Name          Permission  
BITSTransfer  BUILTIN\Administrators AccessAllowed  
  
PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

### [Disable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #43 Unregister-PSSessionConfiguration

### [Unregister-PSSessionConfiguration](#)

#### What can I do with it?

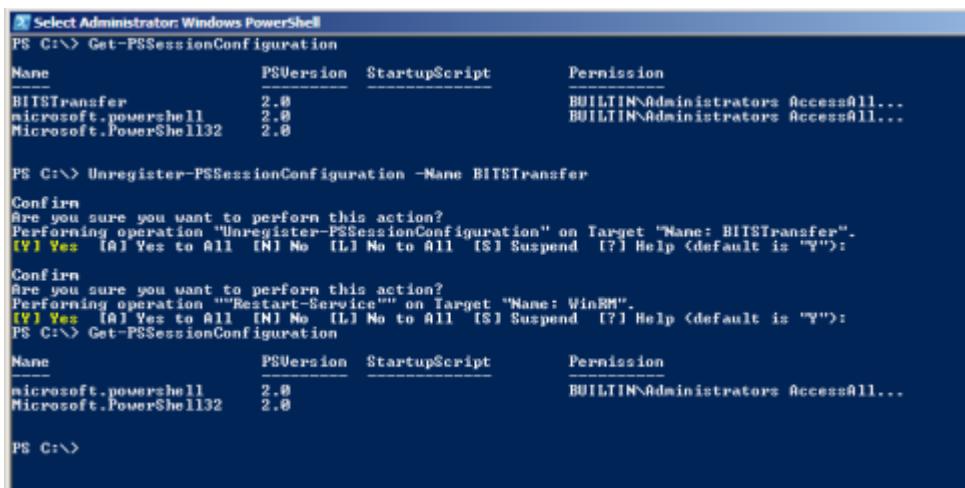
Delete PSSessionConfigurations on the local computer.

#### Example:

View the existing available PSSessionConfigurations with [Get-PSSessionConfiguration](#), remove the **BITSTransfer** configuration and then confirm it has been removed.

```
Get-PSSessionConfiguration  
Unregister-PSSessionConfiguration -Name BITSTransfer  
Get-PSSessionConfiguration
```

You will see that you are prompted to both confirm and the action and the restart of the WinRM service.



```
PS C:\> Get-PSSessionConfiguration  
Name          PSVersion  StartupScript      Permission  
BITSTransfer   2.0         
microsoft.powershell  2.0       BUILTIN\Administrators AccessAll...  
Microsoft.PowerShell32 2.0       BUILTIN\Administrators AccessAll...  
  
PS C:\> Unregister-PSSessionConfiguration -Name BITSTransfer  
Confirm  
Are you sure you want to perform this action?  
Performing operation "Unregister-PSSessionConfiguration" on Target "Name: BITSTransfer".  
(Y) Yes  (A) Yes to All  (N) No  (L) No to All  (S) Suspend  (?) Help <default is "Y">>  
Confirm  
Are you sure you want to perform this action?  
Performing operation "Restart-Service" on Target "Name: WinRM".  
(Y) Yes  (A) Yes to All  (N) No  (L) No to All  (S) Suspend  (?) Help <default is "Y">>  
PS C:\> Get-PSSessionConfiguration  
Name          PSVersion  StartupScript      Permission  
microsoft.powershell  2.0       BUILTIN\Administrators AccessAll...  
Microsoft.PowerShell32 2.0       BUILTIN\Administrators AccessAll...  
  
PS C:\>
```

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

#### Related Cmdlets

[Disable-PSSessionConfiguration](#)

[Enable-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

## #44 Set-WSManQuickConfig

### Set-WSManQuickConfig

#### What can I do with it?

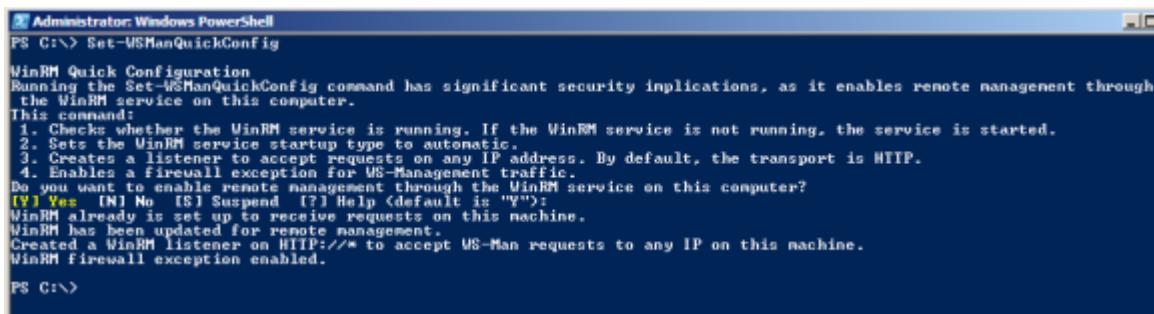
Configure the local computer for use with [WS-Management](#).

#### Example:

Configure the local computer to be enabled for remote management with [WS-Management](#).

### Set-WSManQuickConfig

This will produce output similar to the below; note the command was run on a Windows Server 2008 64-bit system.



```
Administrator: Windows PowerShell
PS C:\> Set-WSManQuickConfig

WinRM Quick Configuration
Running the Set-WSManQuickConfig command has significant security implications, as it enables remote management through the WinRM service on this computer.
This command:
1. Checks whether the WinRM service is running. If the WinRM service is not running, the service is started.
2. Sets the WinRM service startup type to automatic.
3. Creates a listener to accept requests on any IP address. By default, the transport is HTTP.
4. Enables a firewall exception for WS-Management traffic.
Do you want to enable remote management through the WinRM service on this computer?
[Y] Yes [N] No [S] Suspend [?] Help <default is "Y">:
WinRM already is set up to receive requests on this machine.
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.

PS C:\>
```

Set-WSManQuickConfig runs the following tasks:

- Starts the WinRM service if necessary.
- Sets the startup type on the WinRM service to Automatic.
- Creates a listener to accept requests on any IP address.
- Enables a firewall exception for WS-Management communications.

You are prompted to confirm the action.

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Test-WSMan](#)

## #45 Connect-WSMan

### Connect-WSMan

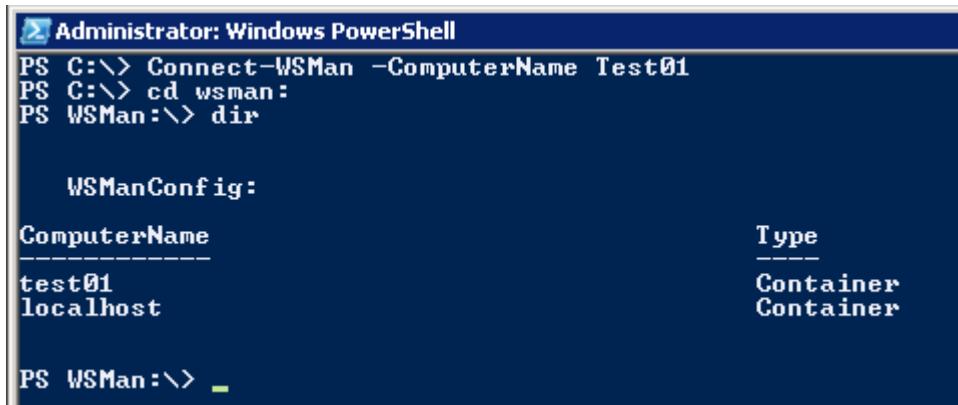
#### What can I do with it?

Create a connection to a remote computer using [WS-Management](#).

#### Example:

Connect to the remote server Test01 using [WS-Management](#). Use the WSMAN provider to examine the WSMAN Shell properties and change the value for **MaxShellsPerUser** to 10.

```
Connect-WSMan -ComputerName Test01
cd wsman:
dir
```



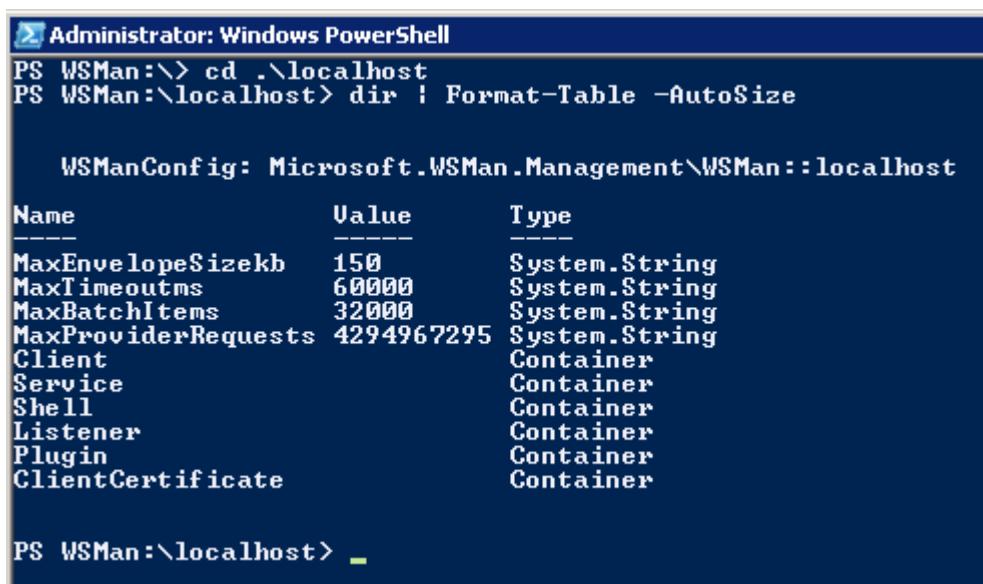
Administrator: Windows PowerShell

```
PS C:\> Connect-WSMan -ComputerName Test01
PS C:\> cd wsman:
PS WSMAN:> dir

    WSMANConfig:
ComputerName          Type
-----            -----
test01                Container
localhost             Container

PS WSMAN:> _
```

```
cd .\localhost
dir | Format-Table -AutoSize
```



Administrator: Windows PowerShell

```
PS WSMAN:> cd .\localhost
PS WSMAN:\localhost> dir | Format-Table -AutoSize

    WSMANConfig: Microsoft.WSMAN.Management\WSMAN::localhost
      Name        Value      Type
      --        ----      ----
MaxEnvelopeSizekb    150      System.String
MaxTimeoutms         60000    System.String
MaxBatchItems        32000    System.String
MaxProviderRequests  4294967295 System.String
Client               Container
Service              Container
Shell                Container
Listener             Container
Plugin               Container
ClientCertificate   Container

PS WSMAN:\localhost> _
```

```
cd Shell
dir | Format-Table -AutoSize
```

```

Administrator: Windows PowerShell
PS WSMAN:\localhost\Shell> cd Shell
PS WSMAN:\localhost\Shell> dir | Format-Table -AutoSize

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Shell

Name          Value   Type
----          ----   ---
AllowRemoteShellAccess true    System.String
IdleTimeout      180000 System.String
MaxConcurrentUsers 5      System.String
MaxShellRunTime   2147483647 System.String
MaxProcessesPerShell 15     System.String
MaxMemoryPerShellMB 150    System.String
MaxShellsPerUser   5      System.String

PS WSMAN:\localhost\Shell> -

```

```

Set-Item -Path MaxShellsPerUser -Value 10
dir | Format-Table -AutoSize

```

```

Administrator: Windows PowerShell
PS WSMAN:\localhost\Shell> Set-Item -Path MaxShellsPerUser -Value 10
PS WSMAN:\localhost\Shell> dir | Format-Table -AutoSize

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Shell

Name          Value   Type
----          ----   ---
AllowRemoteShellAccess true    System.String
IdleTimeout      180000 System.String
MaxConcurrentUsers 5      System.String
MaxShellRunTime   2147483647 System.String
MaxProcessesPerShell 15     System.String
MaxMemoryPerShellMB 150    System.String
MaxShellsPerUser   10     System.String

PS WSMAN:\localhost\Shell> -

```

## How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

### Related Cmdlets

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #46 Test-WSMan

### [Test-WSMan](#)

#### What can I do with it?

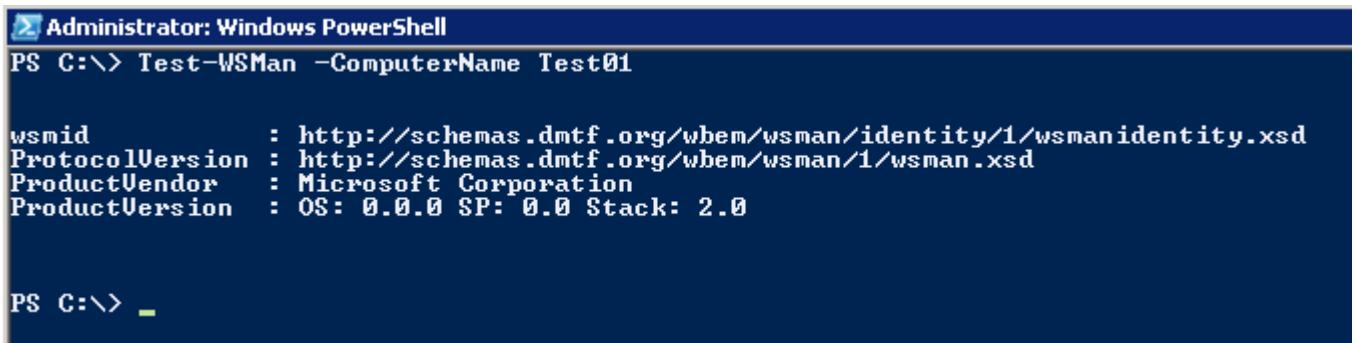
Test whether [WS-Management](#) is available on a computer.

#### Example:

Test whether [WS-Management](#) is available on Test01.

```
Test-WSMan -ComputerName Test01
```

You will notice you receive a response detailing **wsmid**, **ProtocolVersion**, **ProductVendor** and **ProductVersion** if the query is successful.



```
Administrator: Windows PowerShell
PS C:\> Test-WSMan -ComputerName Test01

wsmid      : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 2.0

PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

## #47 Invoke-WSManAction

### [Invoke-WSManAction](#)

#### What can I do with it?

Invoke an action using [WS-Management](#).

#### Examples:

Check the status of the BITS service on Test01, use [WS-Management](#) to stop the service, and then check its status again.

```
Get-Service BITS -ComputerName Test01
Invoke-WSManAction -Action StopService -ResourceURI wmicimv2/Win32_Service
-SelectorSet @{Name="BITS"} -ComputerName Test01 -Authentication Default
Get-Service BITS -ComputerName Test01
```

```
Administrator: Windows PowerShell
PS C:\> Get-Service BITS -ComputerName Test01
Status   Name          DisplayName
Running  BITS          Background Intelligent Transfer Ser...
PS C:\> Invoke-WSManAction -Action StopService -resourceuri wmicimv2/Win32_Service -SelectorSet @Name="BITS" -ComputerName Test01 -Authentication Default
xsi:i  : http://www.w3.org/2001/XMLSchema-instance
p     : http://schemas.microsoft.com/wbem/wsman/1/enum/root/cimv2/Win32_Service
cimn  : http://schemas.dmtf.org/wbem/wscim/1/common
lang   : en-US
ReturnValue : 0

PS C:\> Get-Service BITS -ComputerName Test01
Status   Name          DisplayName
Stopped  BITS          Background Intelligent Transfer Ser...
PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #48 Get-WSManInstance

### Get-WSManInstance

#### What can I do with it?

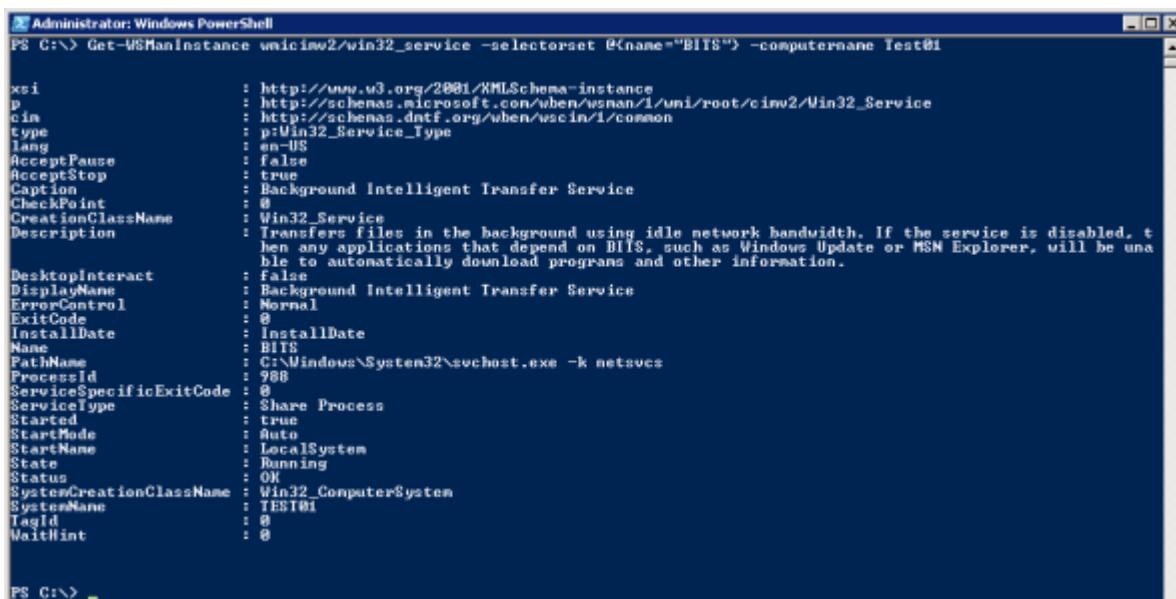
Retrieve an instance of a management resource specified by a URI by using [WS-Management](#).

#### Examples:

Display management information for the BITS service on the remote computer Test01.

```
Get-WSManInstance win32_service  
-SelectorSet @{name="BITS"} -ComputerName Test01
```

Notice that you receive many properties for the BITS service.



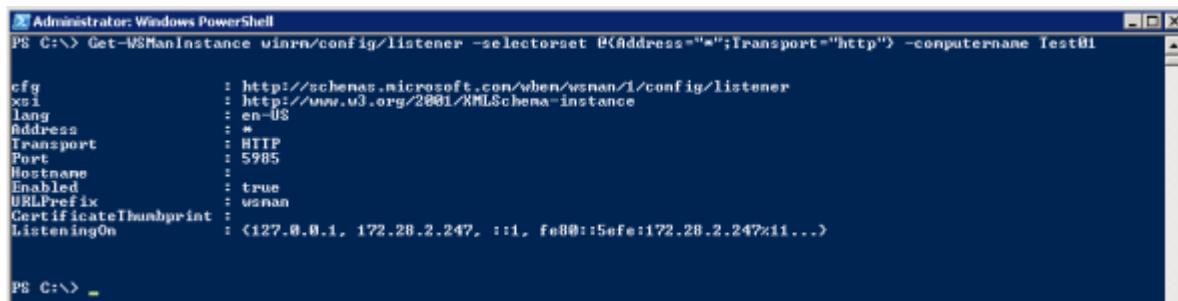
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command run is "Get-WSManInstance win32\_service -SelectorSet @{name="BITS"} -ComputerName Test01". The output displays numerous properties for the BITS service, including:

Property	Value
URI	http://www.w3.org/2001/XMLSchema-instance
Schema	http://schemas.microsoft.com/wbem/wsman/1/uni/root/cimv2/Win32_Service
Type	http://schemas.dntf.org/when/wscin/1/common
Path	p:Win32_Service_Type
Language	en-US
AcceptPause	false
AcceptStop	true
Caption	Background Intelligent Transfer Service
CheckPoint	#
CreationClassName	Win32_Service
Description	Transfers files in the background using idle network bandwidth. If the service is disabled, then any applications that depend on BITS, such as Windows Update or MSM Explorer, will be unable to automatically download programs and other information.
DesktopInteract	false
DisplayName	Background Intelligent Transfer Service
ErrorControl	Normal
ExitCode	0
InstallDate	InstallDate
Name	BITS
PathName	C:\Windows\System32\svchost.exe -k netsvc
ProcessId	988
ServiceSpecificExitCode	0
ServiceType	Share Process
Started	true
StartMode	Auto
StartName	LocalSystem
State	Running
Status	OK
SystemCreationClassName	Win32_ComputerSystem
SystemName	TEST01
TagId	0
WaitHint	0

Display management information for the [WS-Management](#) listener configuration on the remote computer Test01.

```
Get-WSManInstance winrm/config/listener  
-SelectorSet @{}{Address="*";Transport="http"} -ComputerName Test01
```

Notice that you receive a number of properties of the listener.



```
Administrator: Windows PowerShell
PS C:\> Get-WSManInstance winrm/config/listener -SelectorSet @{Address="*";Transport="http"} -ComputerName Test01

cfg          : http://schemas.microsoft.com/wbem/wsman/1/config/listener
xsi          : http://www.w3.org/2001/XMLSchema-instance
lang         : en-US
Address      : *
*:*
Transport   : HTTP
Port         : 5985
Hostname    :
Enabled      : true
URLPrefix   : wsman
CertificateThumbprint : <i27.0.0.1, 172.28.2.247, ::1, fe80::5eFe:172.28.2.247%11...>
ListeningOn : <i27.0.0.1, 172.28.2.247, ::1, fe80::5eFe:172.28.2.247%11...>

PS C:\>
```

## How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

## Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #49 New-WSManInstance

[New-WSManInstance](#)

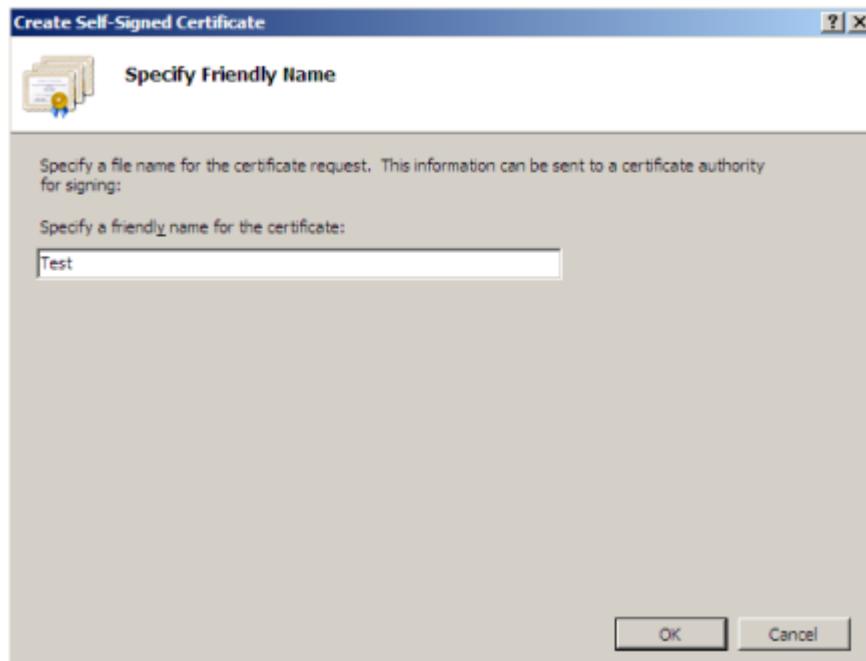
**What can I do with it?**

Create an instance of a management resource for use with [WS-Management](#).

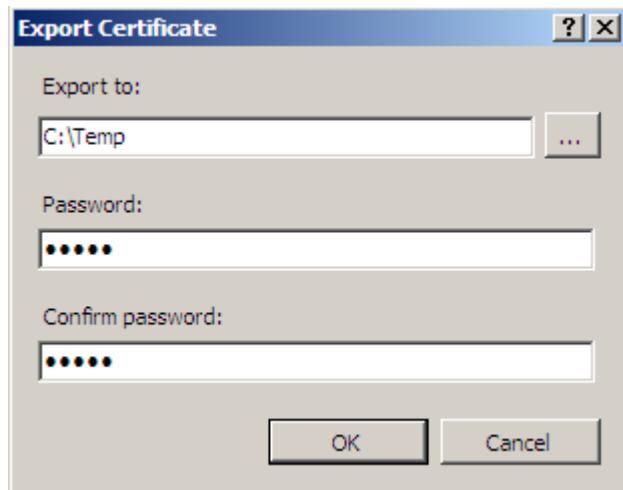
**Example:**

Create an instance of a management resource for use with [WS-Management](#) using HTTPS.

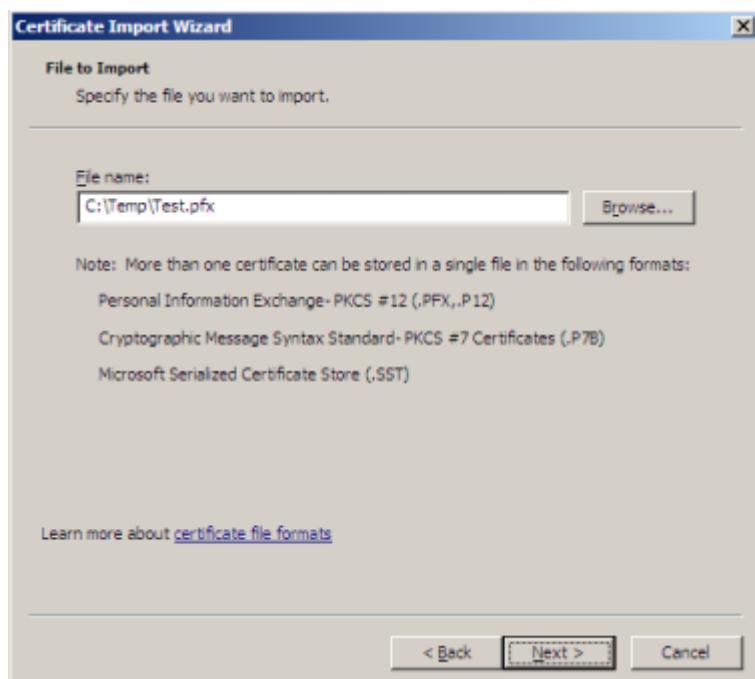
You need to specify a certificate for use with this listener since it is HTTPS. For testing purposes it is possible to create a self-signed certificate within IIS. Open the **Create Self-Signed Certificate Wizard** and enter a name.

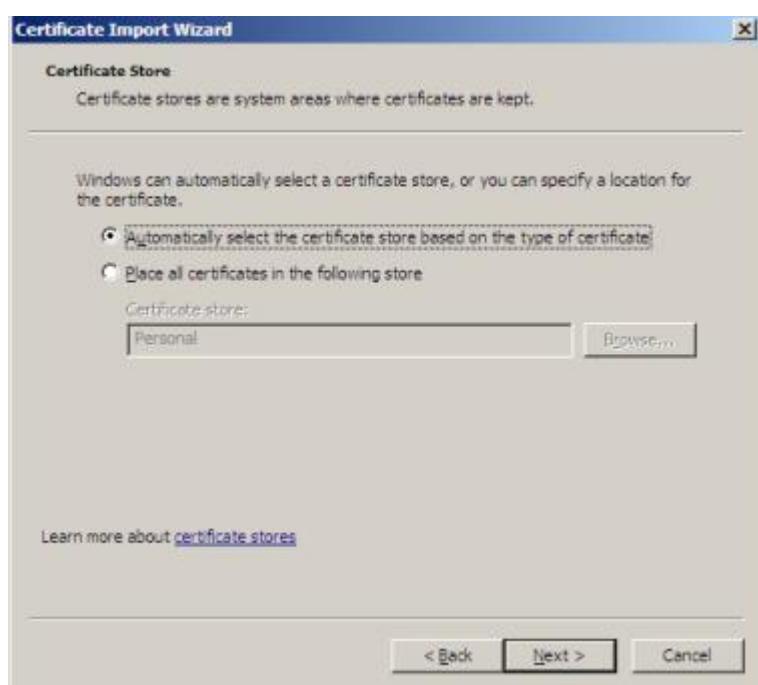


Export it to C:\Temp



Import the pfx file into the Personal Certificate Store





For the New-WSManInstance cmdlet you will require the thumbprint of this certificate, you can find this using PowerShell and the [Certificate Provider](#).

```
Get-Childitem -Path cert:\CurrentUser\My | Format-List  
FriendlyName,Thumbprint
```

The screenshot shows a 'Windows PowerShell' window. The command 'Get-Childitem -Path cert:\CurrentUser\My | Format-List FriendlyName,Thumbprint' is run. The output shows one item: 'FriendlyName : Test' and 'Thumbprint : 01F7EB07A4531750D920CE6A588BFS'. At the bottom, the prompt 'PS C:\> =' is visible.

Creation of the new WSMANInstance using HTTPS is as follows:

```
New-WSManInstance winrm/config/Listener  
-SelectorSet @{Address="*";Transport="HTTPS"}  
-ValueSet  
@{Hostname="Test01";CertificateThumbprint="01F7EB07A4531750D920CE6A588BF5"}
```

```
PS C:\> New-WSManInstance winrm/config/Listener -SelectorSet @{@Address="*";Transport="HTTPS"} -ValueSet @{Hostname="Test01";CertificateThumbprint="01F7EB07A4531750D920CE6A588BF5"}  
wxf : http://schemas.xmlsoap.org/ws/2004/09/transfer  
a : http://schemas.xmlsoap.org/ws/2004/09/addressing  
v : http://schemas.dmtf.org/ubem/usman/1/wsman.xsd  
lang : en-US  
Address : http://schemas.xmlsoap.org/ws/2004/09/addressing/role/anonymous  
ReferenceParameters : ReferenceParameters  
  
PS C:\>
```

You can verify this remotely using the [Get-WSManInstance](#) cmdlet.

```
Get-WSManInstance winrm/config/listener  
-SelectorSet @{@Address="*";Transport="https"} -ComputerName Test01
```

```
PS C:\> Get-WSManInstance winrm/config/listener -selectorset @{@Address="*";Transport="https"} -computername Test01  
cfg : http://schemas.microsoft.com/wbem/wsman/1/config/listener  
xsi : http://www.w3.org/2001/XMLSchema-instance  
lang : en-US  
Address : *  
Transport : HTTPS  
Port : 5986  
Hostname : Test01  
Enabled : true  
URLPrefix : wsman  
CertificateThumbprint : 01F7EB07A4531750D920CE6A588BF5  
ListeningOn : {127.0.0.1, 172.28.2.247, ::1, fe80::5efe:172.28.2.247%11...}  
  
PS C:\>
```

## How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #50 Set-WSManInstance

### Set-WSManInstance

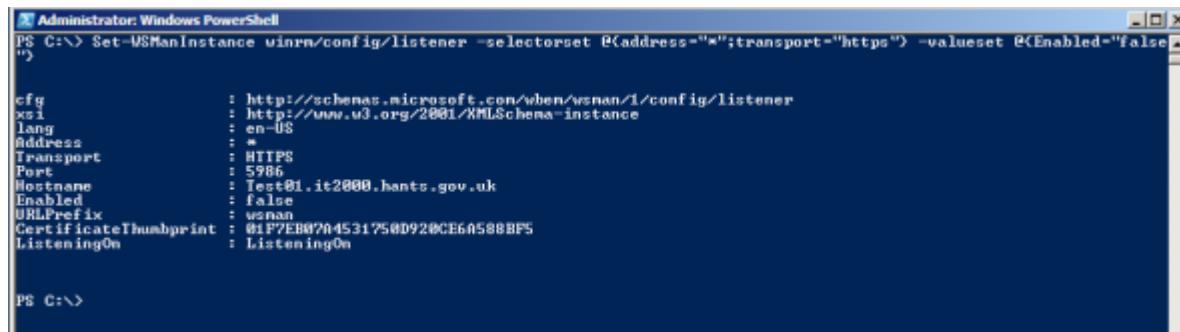
#### What can I do with it?

Change the properties of a management resource for use with [WS-Management](#).

#### Example:

Set the Enabled property of the HTTPS listener created with [New-WSManInstance](#) to **false**, effectively disabling it. **Tip:** watch out for case sensitivity in ValueSet

```
Set-WSManInstance winrm/config/listener  
-SelectorSet @{address="*";transport="https"}  
-ValueSet @{Enabled="false"}
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command executed is `Set-WSManInstance winrm/config/listener -SelectorSet @{address="*";transport="https"} -ValueSet @{Enabled="false"}`. The output displays the configuration settings for the HTTPS listener, including address, transport, port, host name, and the newly set Enabled value of false.

cfg	:	http://schemas.microsoft.com/wbem/wsman/1/config/listener
ps:1	:	http://www.w3.org/2001/XMLSchema-instance
lang	:	en-US
Address	:	*
Transport	:	HTTPS
Port	:	5986
Hostname	:	Test01.it2000.hants.gov.uk
Enabled	:	false
URLPrefix	:	wsman
CertificateThumbprint	:	81F7EB0704531750D920CE6A588BFS
ListeningOn	:	ListeningOn

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #51 Remove-WsManInstance

### Remove-WsManInstance

#### What can I do with it?

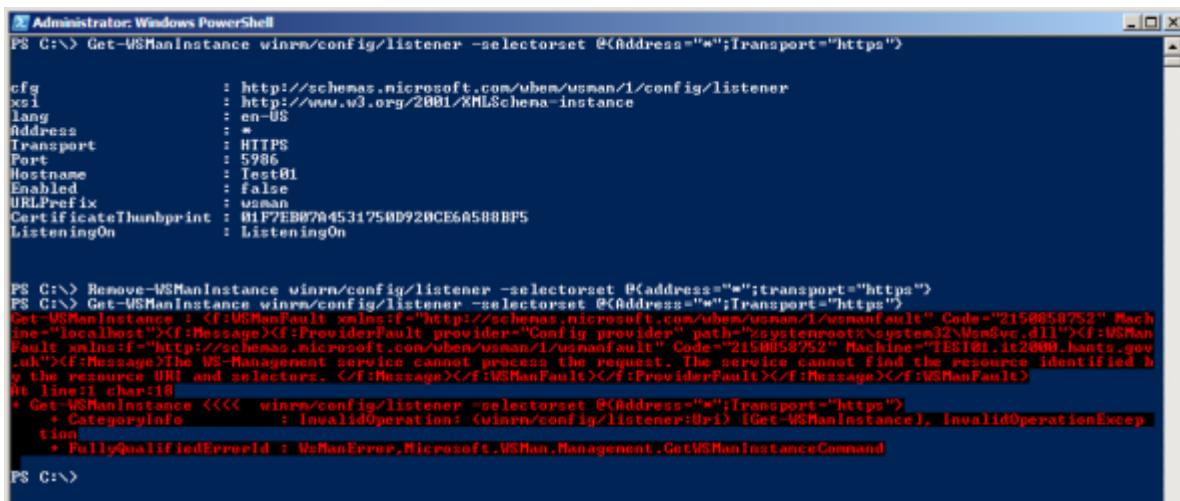
Remove a management resource that has been previously created for use with [WS-Management](#).

#### Example:

Check for existing HTTPS Listeners. Remove the existing HTTPS listener created with [New-WsManInstance](#). Check again to confirm its removal.

```
Get-WsManInstance winrm/config/listener  
-SelectorSet @{Address="*";Transport="https"}  
Remove-WsManInstance winrm/config/listener  
-SelectorSet @{address="*";transport="https"}  
Get-WsManInstance winrm/config/listener  
-SelectorSet @{Address="*";Transport="https"}
```

You will notice that you receive a nasty red error when trying to retrieve it after it has been removed.



```
Administrator: Windows PowerShell  
PS C:\> Get-WsManInstance winrm/config/listener -selectorset @{Address="*";Transport="https"}  
  
cfg : http://schemas.microsoft.com/wbem/wsman/1/config/listener  
xml : http://www.w3.org/2001/XMLSchema-instance  
lang : en-US  
Address : *  
Transport : HTTPS  
Port : 5986  
Hostname : TestB1  
Enabled : false  
URLPrefix : wsman  
CertificateThumbprint : B1P7EBB7A453175BD920CE6A588B5  
ListeningOn : ListeningOn  
  
PS C:\> Remove-WsManInstance winrm/config/listener -selectorset @{}  
PS C:\> Get-WsManInstance winrm/config/listener -selectorset @{}  
Get-WsManInstance : <f:WsManFault xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault" Code="2150050752" Machine="localhost"><f:Message><f:ProviderFault provider="Config provider" path="\\systemroot\\system32\\VmSvc.dll"><f:WsManFault xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault" Code="2150050752" Machine="TESTB1.it2008.hants.gov.uk"><f:Message>The WS-Management service cannot process the request. The service cannot find the resource identified by the resource URI and selectors. </f:Message></f:ProviderFault></f:WsManFault></f:ProviderFault></f:Message></f:WsManFault>  
At line:1 char:18  
+ Get-WsManInstance <<< winrm/config/listener -selectorset @{}  
+ CategoryInfo : InvalidOperation: (winrm/config/listener:Uri) [Get-WsManInstance], InvalidOperationException  
+ FullyQualifiedErrorId : WsManError.Microsoft.WSMan.Management.GetWsManInstanceCommand  
PS C:\>
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

##### [Connect-WsMan](#)

##### [Disable-WsManCredSSP](#)

##### [Disconnect-WsMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #52 New-WSManSessionOption

### New-WSManSessionOption

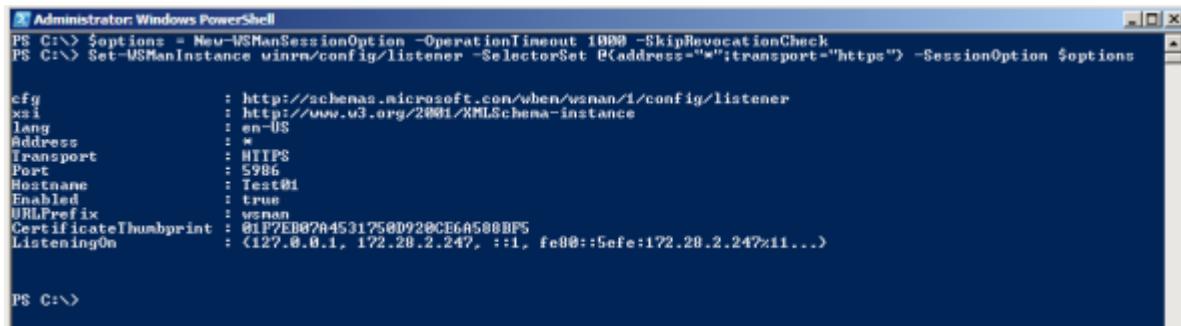
#### What can I do with it?

Create a session option hash table for use with the [WS-Management](#) cmdlets [Get-WSManInstance](#), [Set-WSManInstance](#), [Invoke-WSManAction](#) and [Connect-WSMan](#).

#### Example:

Create a session option hash table for use with the [Set-WSManInstance](#) cmdlet to update the HTTPS listener created with [New-WSManInstance](#).

```
$options = New-WSManSessionOption -OperationTimeout 1000  
-SkipRevocationCheck  
Set-WSManInstance winrm/config/listener  
-SelectorSet @{address="*";transport="https"}  
-SessionOption $options
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $options = New-WSManSessionOption -OperationTimeout 1000 -SkipRevocationCheck  
PS C:\> Set-WSManInstance winrm/config/listener -SelectorSet @{address="*";transport="https"} -SessionOption $options
```

The output displays the configuration of the HTTPS listener:

```
cfg : http://schemas.microsoft.com/wbem/wsman/1/config/listener  
xsi : http://www.w3.org/2001/XMLSchema-instance  
lang : en-US  
Address : *  
Transport : HTTPS  
Port : 5986  
Hostname : Test01  
Enabled : true  
URLPrefix : wsman  
CertificateThumbprint : 01P7EB07A4531750D920CE6A500B5  
ListeningOn : {127.0.0.1, 172.20.2.247, ::1, fe80::5efe:172.20.2.247%11...}
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #53 Enable-WSManCredSSP

### Enable-WSManCredSSP

#### What can I do with it?

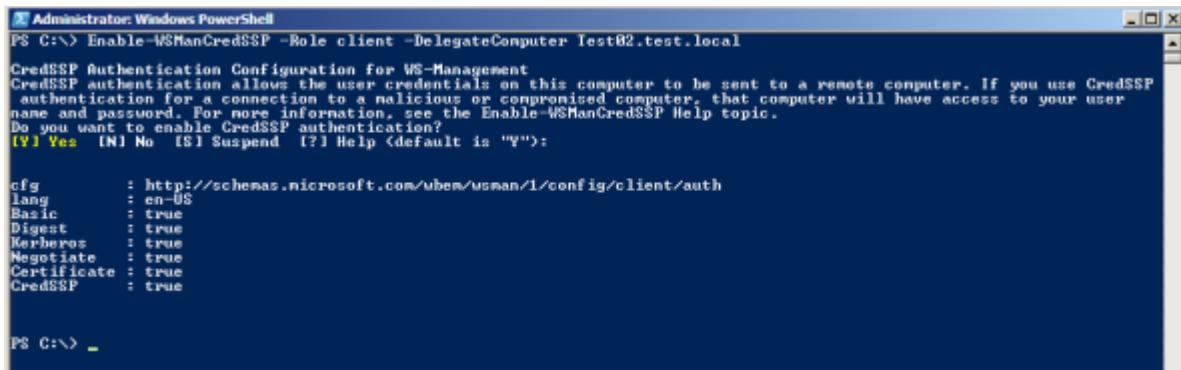
Enable [CredSSP](#) authentication on a computer allowing a user's credentials to be passed to a remote computer for authentication. (Think authentication for background jobs on remote computers.) **Note:** this cmdlet requires running from an elevated PowerShell session.

#### Example:

Enable user's credentials on the local computer to be sent to the remote computer Test02.

```
Enable-WSManCredSSP -Role client -DelegateComputer Test02.test.local
```

You will notice that you are prompted to confirm and given a warning that making this change will allow the remote computer to have access to your username and password.



```
Administrator: Windows PowerShell
PS C:\> Enable-WSManCredSSP -Role client -DelegateComputer Test02.test.local

CredSSP Authentication Configuration for WS-Management
CredSSP authentication allows the user credentials on this computer to be sent to a remote computer. If you use CredSSP
authentication for a connection to a malicious or compromised computer, that computer will have access to your user
name and password. For more information, see the Enable-WSManCredSSP Help topic.
Do you want to enable CredSSP authentication?
[Y] Yes [N] No [S] Suspend [?] Help <default is "Y">:

cfg      : http://schemas.microsoft.com/wbem/wsman/1/config/client/auth
Lang     : en-US
Basic    : true
Digest   : true
Kerberos: true
Negotiate: true
Certificate: true
CredSSP  : true

PS C:\>
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #54 Get-WSManCredSSP

[Get-WSManCredSSP](#)

### What can I do with it?

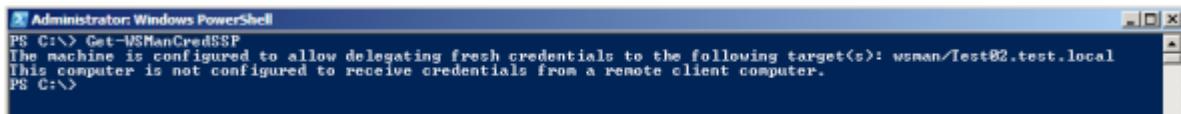
View the [CredSSP](#) configuration on the local computer. **Note:** this cmdlet requires running from an elevated PowerShell session.

#### Example:

View the [CredSSP](#) configuration on the local computer which has previously been enabled for client [CredSSP](#) via [Enable-WSManCredSSP](#).

[Get-WSManCredSSP](#)

You will notice the client part has been enabled, but not the server.



```
Administrator: Windows PowerShell
PS C:\> Get-WSManCredSSP
The machine is configured to allow delegating fresh credentials to the following target(s): wsman/Test02.test.local
This computer is not configured to receive credentials from a remote client computer.
PS C:\>
```

### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #55 Disable-WSManCredSSP

### [Disable-WSManCredSSP](#)

#### What can I do with it?

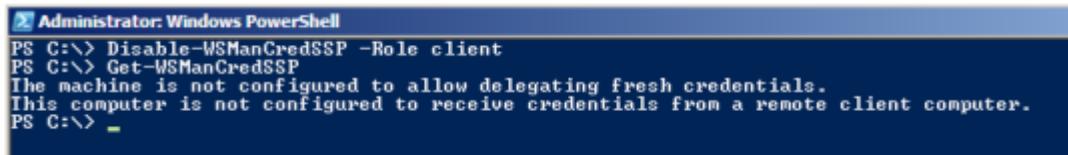
Disable [CredSSP](#) configuration on a computer. **Note:** this cmdlet requires running from an elevated PowerShell session.

#### Example:

Disable the [CredSSP](#) configuration on the local computer which has previously been enabled for client [CredSSP](#) via [Enable-WSManCredSSP](#). Confirm this has been successful with [Get-WSManCredSSP](#).

```
Disable-WSManCredSSP -Role client  
Get-WSManCredSSP
```

You will notice that the computer is no longer configured for [CredSSP](#) authentication.



```
Administrator: Windows PowerShell  
PS C:\> Disable-WSManCredSSP -Role client  
PS C:\> Get-WSManCredSSP  
The machine is not configured to allow delegating fresh credentials.  
This computer is not configured to receive credentials from a remote client computer.  
PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disconnect-WSMan](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #56 Disconnect-WSMan

[Disconnect-WSMan](#)

### What can I do with it?

Disconnect a connection previously made to a remote computer using [WS-Management](#) with the [Connect-WSMan](#) cmdlet.

#### Example:

Disconnect from the remote server Test01 using [WS-Management](#) .

```
Disconnect-WSMan -ComputerName Test01
```

### How could I have done this in PowerShell 1.0?

Support for the use of [WS-Management](#) in PowerShell is provided as part of the 2.0 release.

#### Related Cmdlets

[Connect-WSMan](#)

[Disable-WSManCredSSP](#)

[Enable-PSRemoting](#)

[Enable-WSManCredSSP](#)

[Get-WSManCredSSP](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-PSSession](#)

[New-WSManInstance](#)

[New-WSManSessionOption](#)

[Remove-WSManInstance](#)

[Set-WSManInstance](#)

[Set-WSManQuickConfig](#)

[Test-WSMan](#)

## #57 Import-PSSession

### Import-PSSession

#### What can I do with it?

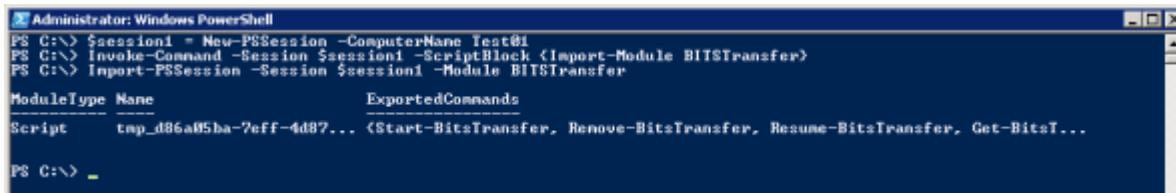
Import commands from a Remote PowerShell session into the current session, for instance from a remote session on another computer.

#### Example:

Establish a remote session with Test01 using [New-PSSession](#). Use [Invoke-Command](#) to initiate the use of the BITSTransfer module. Use Import-PSSession to make the contents of the BITSTransfer module available in the local session even though the BITSTransfer module has not been imported on the local computer.

**Note:** Technically I could just have imported the BITSTransfer module on the local machine; however, this example is to demonstrate that potentially any module could be brought across to the local session.

```
$session1 = New-PSSession -ComputerName Test01
Invoke-Command -Session $session1 -ScriptBlock {Import-Module BITSTransfer}
Import-PSSession -Session $session1 -Module BITSTransfer
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history shows:

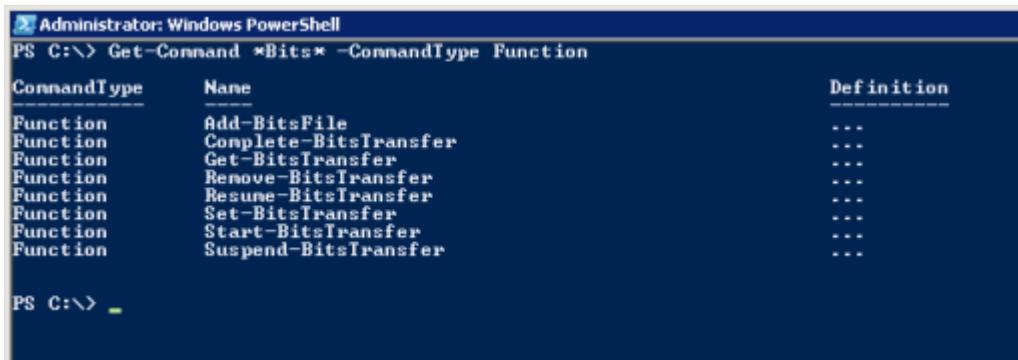
```
PS C:\> $session1 = New-PSSession -ComputerName Test01
PS C:\> Invoke-Command -Session $session1 -ScriptBlock {Import-Module BITSTransfer}
PS C:\> Import-PSSession -Session $session1 -Module BITSTransfer
```

The output shows the imported module information:

ModuleType	Name	ExportedCommands
Script	mp_d86a05ba-2eff-4d87... (Start-BitsTransfer, Remove-BitsTransfer, Resume-BitsTransfer, Get-BitsT...)	

Confirm the contents of the BITSTransfer module is now available in the local session.

```
Get-Command *Bits* - CommandType Function
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history shows:

```
PS C:\> Get-Command *Bits* - CommandType Function
```

The output is a table showing the functions defined in the Bits module:

CommandType	Name	Definition
Function	Add-BitsFile	...
Function	Complete-BitsTransfer	...
Function	Get-BitsTransfer	...
Function	Remove-BitsTransfer	...
Function	Resume-BitsTransfer	...
Function	Set-BitsTransfer	...
Function	Start-BitsTransfer	...
Function	Suspend-BitsTransfer	...

#### Extras:

Ravikanth Chaganti has an excellent post covering this cmdlet in more detail [here](#).

#### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0, you would have needed to use Remote Desktop to run an interactive session on a remote server.

## **Related Cmdlets**

[New-PSSession](#)

[Export-PSSession](#)

## #58 Export-PSSession

### Export-PSSession

#### What can I do with it?

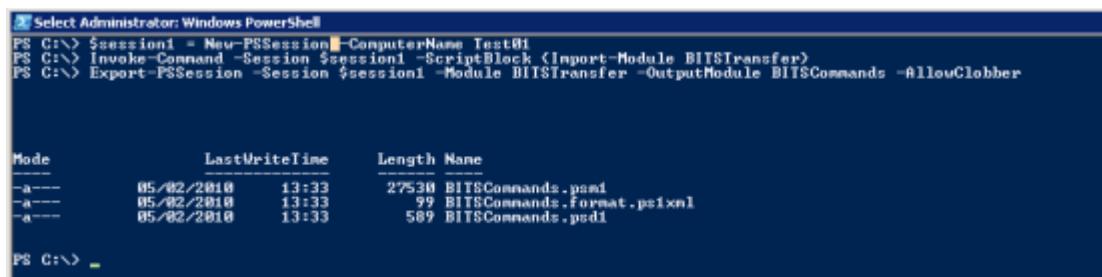
Export commands from a remote PowerShell session into a module saved on the local system.

#### Example:

Establish a remote session with Test01 using [New-PSSession](#). Use [Invoke-Command](#) to initiate the use of the BITSTransfer module. Export the commands from the BITSTransfer module into a module saved on the local system and called BITSCommands.

```
$session1 = New-PSSession -ComputerName Test01
Invoke-Command -Session $session1 -ScriptBlock {Import-Module BITSTransfer}
Export-PSSession -Session $session1 -Module BITSTransfer
-OutputModule BITSCommands -AllowClobber
```

The exported files are stored within your PowerShell profile folder.



A screenshot of a Windows PowerShell window titled "Select Administrator: Windows PowerShell". The command history shows:

```
PS C:\> $session1 = New-PSSession -ComputerName Test01
PS C:\> Invoke-Command -Session $session1 -ScriptBlock {Import-Module BITSTransfer}
PS C:\> Export-PSSession -Session $session1 -Module BITSTransfer
-OutputModule BITSCommands -AllowClobber
```

Below the command history, a table lists the contents of the BITSCommands module:

Mode	LastWriteTime	Length	Name
-a---	05/02/2010	13:33	27530 BITSCommands.psm1
-a---	05/02/2010	13:33	99 BITSCommands.Format.ps1xml
-a---	05/02/2010	13:33	589 BITSCommands.psd1

The contents of the BITSCommands.psd1 file are below:

```
1
2 <#
3 # Implicit remoting module
4 # generated on 05/02/2010 13:33:16
5 # by Export-PSSession cmdlet
6 # invoked with the following command line: Export-PSSession -Session $session1 -Module BITSTransfer -OutputModule BITSCommands -AllowClobber
7 >
8
9 <#
10 GUID = '7c395776-9cdd-90d1-e4e4-b79d3ce4c740'
11 Description = 'Implicit remoting for http://test01/wssession'
12 ModuleToProcess = @('BITSCommands.psm1')
13 FormatsToProcess = @('BITSCommands.Format.ps1xml')
14
15 ModuleVersion = '1.0'
16
17 PrivateData = {
18     ImplicitRemoting = $true
19 }
20
21
```

You could make use of this module at a later date with:

```
Import-Module BITSCommands
```

#### Extras:

Ravikanth Chaganti has an excellent post covering this cmdlet in more detail [here](#).

## **How could I have done this in PowerShell 1.0?**

Remoting did not exist in PowerShell 1.0, you would have needed to use Remote Desktop to run an interactive session on a remote server.

### **Related Cmdlets**

[New-PSSession](#)

[Import-PSSession](#)

## #59 Set-PSBreakpoint

### Set-PSBreakpoint

#### What can I do with it?

Carry out debugging by setting a breakpoint based on a condition such as line number, command or variable.

#### Examples:

Set a breakpoint at line 3 in the script C:\Bowling.ps1 (This is an example script taken from the [2008 Scripting Games](#). During the execution of the script the variable \$iPoints is frequently incremented to a new value) Then run the script to utilise the breakpoint.

```
Set-PSBreakpoint -Script Bowling.ps1 -Line 3
```

You receive confirmation of the breakpoint set:

```
Administrator: Windows PowerShell
PS C:\> Set-PSBreakpoint -Script Bowling.ps1 -Line 3
ID Script          Line Command          Variable          Action
-- -- -- -- -- -- --
 0 Bowling.ps1      3
PS C:\> _
```

Now when you run the script, you are informed at what point we have stopped at and the current value of \$iPoints. Typing **Exit** will leave the debugging mode.

```
Administrator: Windows PowerShell
PS C:\> .\Bowling.ps1
Entering debug mode. Use h or ? for help.
Hit Line breakpoint on 'C:\Bowling.ps1:3'
Bowling.ps1:3  $iPoints = 0
[DBG]: PS C:\>>> Exit
107
PS C:\> _
```

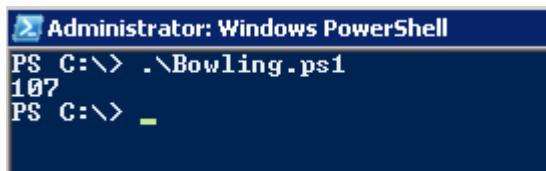
For the next example set a breakpoint based on the variable \$iPoints and carry out an action to save the value of \$iPoints at that point in time into the file C:\log.txt.

```
Set-PSBreakpoint -Script .\Bowling.ps1 -Variable iPoints
-Action {Out-File log.txt -Append -InputObject $iPoints}
```

This time the confirmation shows both a Variable and an Action have been set as part of the Breakpoint.

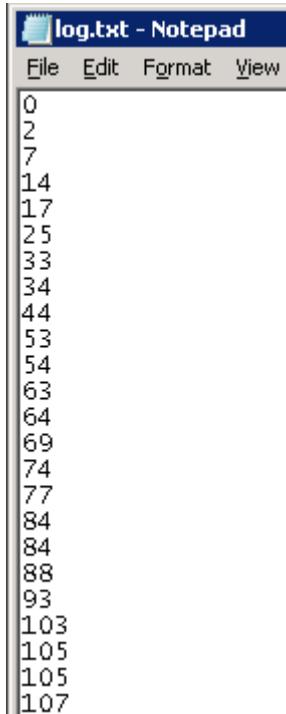
```
Administrator: Windows PowerShell
PS C:\> Set-PSBreakpoint -Script .\Bowling.ps1 -Variable iPoints -Action {Out-File log.txt -Append -InputObject $iPoints}
ID Script          Line Command          Variable          Action
-- -- -- -- -- -- --
 1 Bowling.ps1      3               iPoints        Out-File log.txt -Appen...
```

Running the script does not bring up the interactive debugger this time.



```
Administrator: Windows PowerShell
PS C:\> .\Bowling.ps1
107
PS C:\> _
```

However, the log.txt file is created and its contents show the value of \$iPoints each time it is referenced in the script.



### How could I have done this in PowerShell 1.0?

Setting breakpoints did not exist in PowerShell 1.0, however most scripting IDEs ship with debugging features.

### Related Cmdlets

[Get-PSBreakpoint](#)

[Enable-PSBreakpoint](#)

[Disable-PSBreakpoint](#)

[Remove-PSBreakpoint](#)

## #60 Get-PSBreakpoint

### [Get-PSBreakpoint](#)

#### What can I do with it?

Retrieve debugging breakpoints that have been set with [Set-PSBreakpoint](#).

#### Examples:

Retrieve all current breakpoints.

#### [Get-PSBreakpoint](#)

Notice the different options which have been set on the breakpoints.

ID	Script	Line	Command	Variable	Action
0	Bowling.ps1	3		iPoints	
1	Bowling.ps1				Out-File log.txt -Appen...

Retrieve only breakpoints which have been set using the **Variable** parameter.

#### [Get-PSBreakpoint -Type Variable](#)

Notice only one breakpoint is returned this time.

ID	Script	Line	Command	Variable	Action
1	Bowling.ps1			iPoints	Out-File log.txt -Appen...

#### How could I have done this in PowerShell 1.0?

Setting breakpoints did not exist in PowerShell 1.0, however most scripting IDEs ship with debugging features.

#### Related Cmdlets

##### [Enable-PSBreakpoint](#)

##### [Disable-PSBreakpoint](#)

##### [Remove-PSBreakpoint](#)

##### [Set-PSBreakpoint](#)

## #61 Disable-PSBreakpoint

### [Disable-PSBreakpoint](#)

#### What can I do with it?

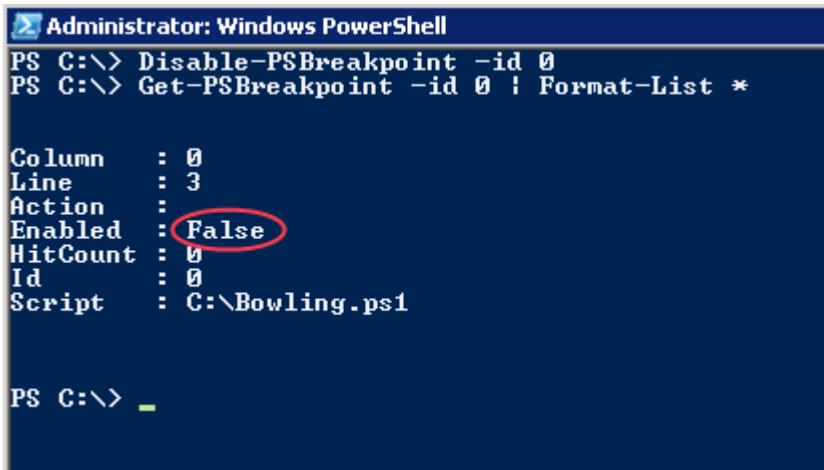
Disable debugging breakpoints that have been set with [Set-PSBreakpoint](#).

#### Example:

Disable the breakpoint with ID 0 and then check its properties to confirm it has been disabled.

```
Disable-PSBreakpoint -Id 0
Get-PSBreakpoint -Id 0 | Format-List *
```

You will notice that the **Enabled** property is set to **False**.



```
Administrator: Windows PowerShell
PS C:\> Disable-PSBreakpoint -id 0
PS C:\> Get-PSBreakpoint -id 0 | Format-List *

Column      : 0
Line        : 3
Action      :
Enabled     : False
HitCount    : 0
Id          : 0
Script      : C:\Bowling.ps1

PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

Setting breakpoints did not exist in PowerShell 1.0, however most scripting IDEs ship with debugging features.

#### Related Cmdlets

[Get-PSBreakpoint](#)

[Enable-PSBreakpoint](#)

[Remove-PSBreakpoint](#)

[Set-PSBreakpoint](#)

## #62 Enable-PSBreakpoint

[Enable-PSBreakpoint](#)

### What can I do with it?

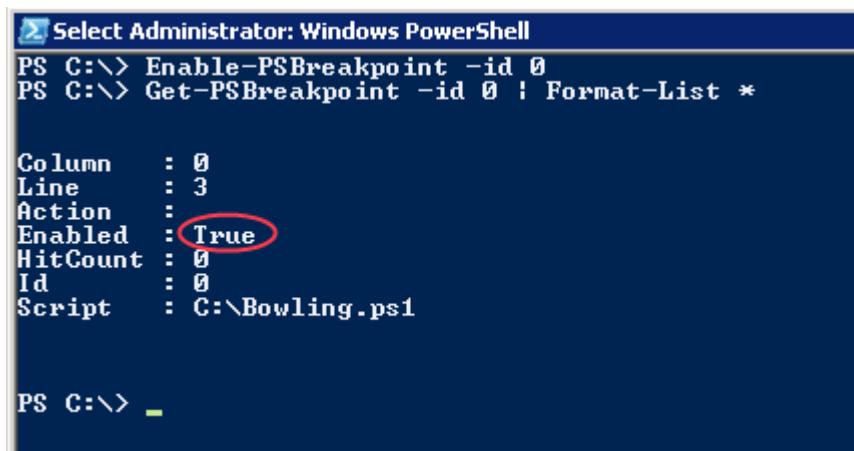
Re-enable debugging breakpoints that have been disabled with [Disable-PSBreakpoint](#).

#### Example:

Re-enable breakpoint with ID 0 and then check its properties to confirm it has been enabled.

```
Enable-PSBreakpoint -Id 0
Get-PSBreakpoint -Id 0 | Format-List *
```

You will notice that the **Enabled** property is set to **True**.



The screenshot shows a Windows PowerShell window titled "Select Administrator: Windows PowerShell". The command "Enable-PSBreakpoint -Id 0" is run, followed by "Get-PSBreakpoint -Id 0 | Format-List \*". The resulting output is:

Column	:	0
Line	:	3
Action	:	
Enabled	:	True
HitCount	:	0
Id	:	0
Script	:	C:\Bowling.ps1

A red oval highlights the "Enabled" value "True".

### How could I have done this in PowerShell 1.0?

Setting breakpoints did not exist in PowerShell 1.0, however most scripting IDEs ship with debugging features.

### Related Cmdlets

[Get-PSBreakpoint](#)

[Disable-PSBreakpoint](#)

[Remove-PSBreakpoint](#)

[Set-PSBreakpoint](#)

## #63 Remove-PSBreakpoint

[Remove-PSBreakpoint](#)

**What can I do with it?**

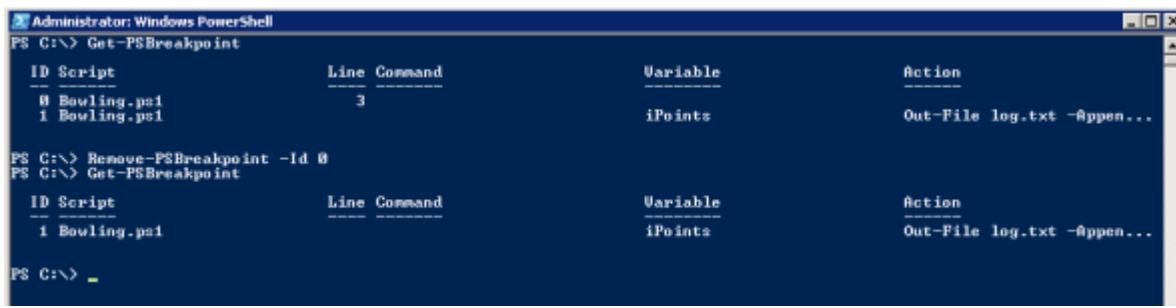
Remove debugging breakpoints that have been set with [Set-PSBreakpoint](#).

**Examples:**

Check existing breakpoints and remove the breakpoint with ID 0.

```
Get-PSBreakpoint  
Remove-PSBreakpoint -Id 0
```

Confirmation that breakpoint with ID 0 has been removed.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays two tables of breakpoint information. The first table, after running "Get-PSBreakpoint", shows a single breakpoint with ID 0, script "Bowling.ps1", line 3, variable "iPoints", and action "Out-File log.txt -Appen...". The second table, after running "Remove-PSBreakpoint -Id 0" followed by "Get-PSBreakpoint", shows no breakpoints.

ID	Script	Line	Command	Variable	Action
0	Bowling.ps1	3		iPoints	Out-File log.txt -Appen...

ID	Script	Line	Command	Variable	Action
1	Bowling.ps1			iPoints	Out-File log.txt -Appen...

Check existing breakpoints and remove all of them.

```
Get-PSBreakpoint  
Get-PSBreakpoint | Remove-PSBreakpoint
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays two tables of breakpoint information. The first table, after running "Get-PSBreakpoint", shows a single breakpoint with ID 0, script "Bowling.ps1", line 3, variable "iPoints", and action "Out-File log.txt -Appen...". The second table, after running "Get-PSBreakpoint | Remove-PSBreakpoint" followed by "Get-PSBreakpoint", shows no breakpoints.

ID	Script	Line	Command	Variable	Action
0	Bowling.ps1	3		iPoints	Out-File log.txt -Appen...

ID	Script	Line	Command	Variable	Action
1	Bowling.ps1			iPoints	Out-File log.txt -Appen...

**How could I have done this in PowerShell 1.0?**

Setting breakpoints did not exist in PowerShell 1.0, however most scripting IDEs ship with debugging features.

**Related Cmdlets**

[Get-PSBreakpoint](#)

[Enable-PSBreakpoint](#)

[Disable-PSBreakpoint](#)

[Set-PSBreakpoint](#)

## #64 Clear-History

### [Clear-History](#)

#### What can I do with it?

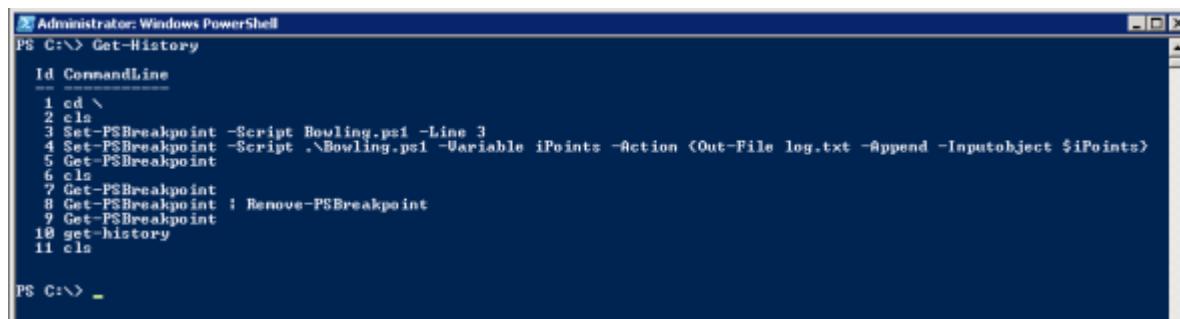
Remove commands from the history of those entered in the current session. PowerShell has two places where a history of the commands you have entered are kept. Within the console you can use **F7** to view them and **Alt-F7** to clear that list. There are also some cmdlets for managing PowerShell history, such as [Get-History](#) and the new [Clear-History](#).

#### Example:

Check current history. Then remove any commands from the history which contain the string **set**.

```
Get-History  
Clear-History -CommandLine *set*
```

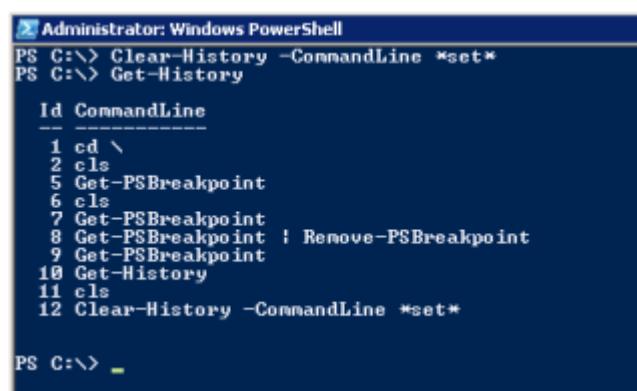
The initial history is as below:



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command prompt is PS C:\>. The output shows the following command history:

```
PS C:\> Get-History  
Id CommandLine  
---  
1 cd \  
2 cls  
3 Set-PSBreakpoint -Script Bowling.ps1 -Line 3  
4 Set-PSBreakpoint -Script .\Bowling.ps1 -Variable iPoints -Action Out-File log.txt -Append -InputObject $iPoints  
5 Get-PSBreakpoint  
6 cls  
7 Get-PSBreakpoint  
8 Get-PSBreakpoint : Remove-PSBreakpoint  
9 Get-PSBreakpoint  
10 get-history  
11 cls  
PS C:\>
```

Now we remove any commands from the history which contain the string **set**. You can see they have been removed and the others remain.



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command prompt is PS C:\>. The output shows the following command history after running `Clear-History -CommandLine *set*`:

```
PS C:\> Clear-History -CommandLine *set*  
PS C:\> Get-History  
Id CommandLine  
---  
1 cd \  
2 cls  
5 Get-PSBreakpoint  
6 cls  
7 Get-PSBreakpoint  
8 Get-PSBreakpoint : Remove-PSBreakpoint  
9 Get-PSBreakpoint  
10 Get-History  
11 cls  
12 Clear-History -CommandLine *set*
```

#### How could I have done this in PowerShell 1.0?

Andrew Watt explains how you can clear history in PowerShell 1.0 on this [forum post](#).

From a new session, set the preference \$MaximumHistoryCount variable to 1, then get the current history and export it to XML.

```
$MaximumHistoryCount = 1  
Get-History | Export-Clixml "History.xml"
```

Edit the XML document, remove the text between <S> and replace it with "No commands have been entered"

```
- <Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">  
- <Obj RefId="0">  
- <TN RefId="0">  
  <T>Microsoft.PowerShell.Commands.HistoryInfo</T>  
  <T>System.Object</T>  
</TN>  
  <ToString>$MaximumHistoryCount = 1</ToString>  
- <Props>  
  <I64 N="Id">2</I64>  
  <S N="CommandLine">No commands have been entered</S>  
- <Obj N="ExecutionStatus" RefId="1">
```

Create a script called **Empty-History.ps1** containing the below:

```
function global:Empty-History{  
$MaximumHistoryCount = 1  
Import-Clixml "History.xml" | Add-History  
}
```

Now dot source the script and use the **Empty-History** function to clear your history.

## Related Cmdlets

[Get-History](#)

[Add-History](#)

[Invoke-History](#)

## #65 New-EventLog

### [New-EventLog](#)

#### What can I do with it?

Create a custom Event Log.

#### Example:

Create a custom Event Log named **App1** with an event source of **AppEvent**. Use the `Get-EventLog` cmdlet to confirm it has been created. **Tip:** `New-EventLog` requires a PowerShell session with elevated privileges.

```
New-EventLog -LogName App1 -Source AppEvent  
Get-EventLog -List
```

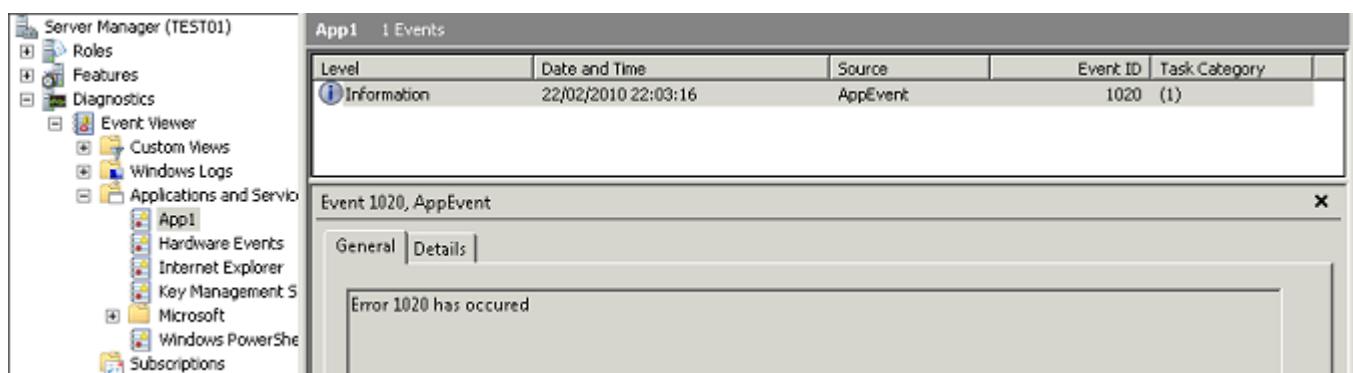
You can see that the **App1** Event Log has been created.

PS C:\> New-EventLog -LogName App1 -Source AppEvent	PS C:\> Get-EventLog -List			
Max<(K)	Retain	OverflowAction	Entries	Log
512	7	OverwriteOlder	0	App1
20,480	0	OverwriteAsNeeded	778	Application
20,480	0	OverwriteAsNeeded	0	HardwareEvents
512	7	OverwriteOlder	0	Internet Explorer
20,480	0	OverwriteAsNeeded	0	Key Management Service
20,480	0	OverwriteAsNeeded	7,435	Security
20,480	0	OverwriteAsNeeded	10,597	System
15,360	0	OverwriteAsNeeded	295	Windows PowerShell

You can create entries in this log using the [Write-EventLog](#) cmdlet, e.g.

```
Write-EventLog -LogName App1 -Source AppEvent -ID 1020  
-Message "Error 1020 has occurred"
```

Here's confirmation in Event Viewer that the App1 Event Log exists and we have created the above entry in it.



#### How could I have done this in PowerShell 1.0?

You could have used the .NET [System.Diagnostics.EventLog class](#) and the [CreateEventSource method](#) to create a custom event log.

```
$LogDetails =  
New-Object System.Diagnostics.EventSourceCreationData "AppEvent", "App1"  
[System.Diagnostics.EventLog]::CreateEventSource($LogDetails)
```

## Related Cmdlets

[Clear-EventLog](#)

[Get-EventLog](#)

[Limit-EventLog](#)

[Remove-EventLog](#)

[Show-EventLog](#)

[Write-EventLog](#)

[Get-WinEvent](#)

## #66 Limit-EventLog

### Limit-EventLog

#### What can I do with it?

Set the size and age properties of an Event Log.

#### Example:

Set the following properties on the **Application Log** on the remote computer **Test01**:

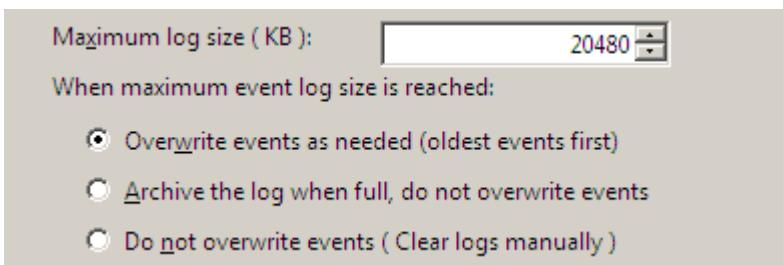
Maximum Size = 5MB

OverflowAction = DoNotOverWrite

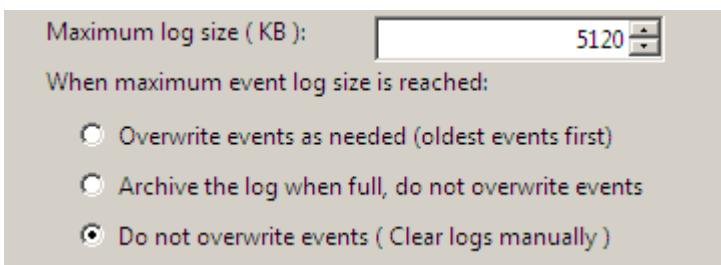
```
Limit-EventLog -ComputerName Test01 -LogName Application  
-MaximumSize 5MB -OverflowAction DoNotOverWrite
```



#### Before:



#### After:



#### How could I have done this in PowerShell 1.0?

You could use WMI to set properties on an event log. For example to set the **MaxFileSize** of the **Application Log** to **5MB** use the below. (Thanks to [Richard Siddaway for the tip](#) that you need to use psbase to save the changes, just **Put()** doesn't work.)

```
$EventLog = Get-WmiObject -Class Win32_NTEventLogFile  
-Filter "LogFileName = 'Application'"
```

```
$EventLog.MaxValueSize = 5242880  
$EventLog.psbase.Put()
```

## Related Cmdlets

[Clear-EventLog](#)

[Get-EventLog](#)

[New-EventLog](#)

[Remove-EventLog](#)

[Show-EventLog](#)

[Write-EventLog](#)

[Get-WinEvent](#)

## #67 Remove-EventLog

### Remove-EventLog

#### What can I do with it?

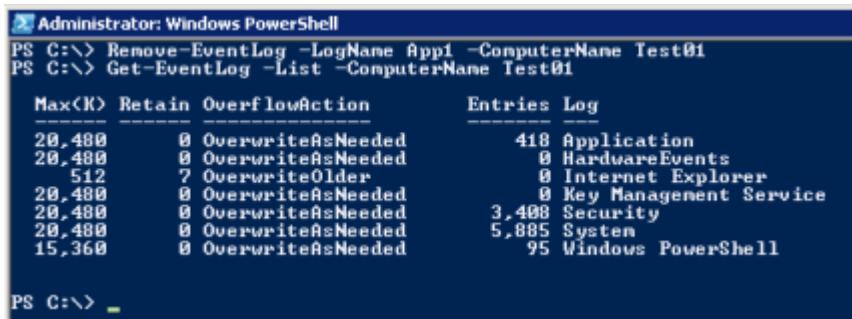
Remove an Event Log.

#### Example:

Remove the Event Log named **App1** on the remote computer **Test01**. Confirm it has been removed with **Get-EventLog**.

```
Remove-EventLog -LogName App1 -ComputerName Test01
Get-EventLog -List -ComputerName Test01
```

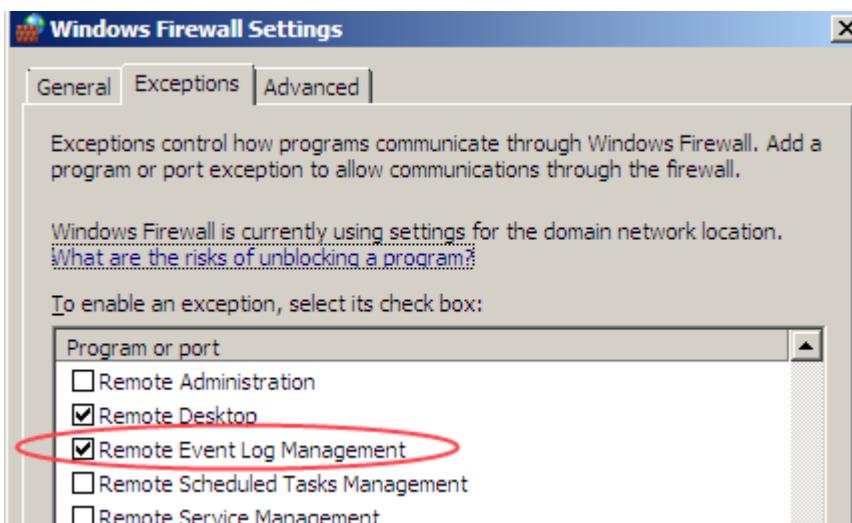
Confirmation that the **App1** Event Log has been removed.



```
Administrator: Windows PowerShell
PS C:\> Remove-EventLog -LogName App1 -ComputerName Test01
PS C:\> Get-EventLog -List -ComputerName Test01
Max(K) Retain OverflowAction      Entries Log
----- -- -- -- -- 
20,400 0 OverwriteAsNeeded      418 Application
20,400 0 OverwriteAsNeeded      0 HardwareEvents
512    7 OverwriteOlder        0 Internet Explorer
20,400 0 OverwriteAsNeeded      0 Key Management Service
20,400 0 OverwriteAsNeeded      3,408 Security
20,400 0 OverwriteAsNeeded      5,885 System
15,360 0 OverwriteAsNeeded      95 Windows PowerShell

PS C:\> _
```

**Note:** To perform this task remotely you will need to ensure that **Remote Event Log Management** has been added as an Exception in Windows Firewall.



#### How could I have done this in PowerShell 1.0?

You could have used the .NET [System.Diagnostics.Eventlog](#) class and the Delete method to delete an event log.

```
[system.diagnostics.eventlog]::Delete("App1")
```

## Related Cmdlets

[Clear-EventLog](#)

[Get-EventLog](#)

[Limit-EventLog](#)

[New-EventLog](#)

[Show-EventLog](#)

[Write-EventLog](#)

[Get-WinEvent](#)

## #68 Show-EventLog

[Show-EventLog](#)

**What can I do with it?**

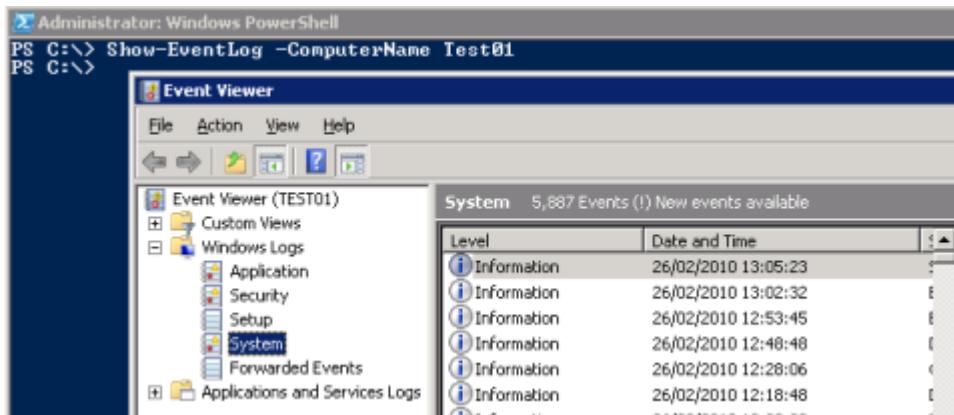
Open Event Viewer on a local or remote computer.

**Example:**

Open Event Viewer on the remote computer **Test01**.

`Show-EventLog -ComputerName Test01`

You will see that Event Viewer on the remote computer **Test01** opens on the local machine.



**How could I have done this in PowerShell 1.0?**

You could have typed the executable **Eventvwr** to open Event Viewer on a local computer.  
To view it on the remote computer **Test01** use:

`eventvwr \\Test01`

**Related Cmdlets**

[Clear-EventLog](#)

[Get-EventLog](#)

[Limit-EventLog](#)

[New-EventLog](#)

[Remove-EventLog](#)

[Write-EventLog](#)

[Get-WinEvent](#)

## #69 Get-WinEvent

### [Get-WinEvent](#)

#### What can I do with it?

Retrieve items from Event Logs including event logs generated by the Windows Event Log technology, new since Windows Vista / 2008 Server, in addition to the classic System, Security and Application Logs. **Note:** it requires .NET Framework 3.5 or later installed.

#### Examples:

Retrieve events from the **Setup** Event Log.

```
Get-WinEvent -LogName Setup
```

You'll see the typical information you would normally view in Event Viewer.

Administrator: Windows PowerShell		
TimeCreated	ProviderName	Id Message
01/03/2010 17:37:41	Microsoft-Windows-ServerMa...	1610 Installation succeeded. ...
01/03/2010 17:37:39	Microsoft-Windows-Servicing	9 Selectable update NetFx3 e...
01/03/2010 17:37:25	Microsoft-Windows-Servicing	7 Initiating changes to turn...

Get-WinEvent includes the **-FilterHashTable** parameter which allows you to filter results at source rather than pulling back all the events and then piping them through to Where-Object to perform filtering, so much more efficient.

Retrieve events from the **System** Event Log only where the **Event ID** is **10148**.

```
Get-WinEvent -FilterHashtable @{Logname='System'; ID=10148}
```

You will see that only the events with **ID 10148** are returned.

Administrator: Windows PowerShell		
TimeCreated	ProviderName	Id Message
01/03/2010 17:37:45	Microsoft-Windows-WinRM	10148 The WinRM service is liste...
01/03/2010 17:39:31	Microsoft-Windows-WinRM	10148 The WinRM service is liste...

#### How could I have done this in PowerShell 1.0?

You could have used the Get-EventLog cmdlet, however, it is not able to retrieve information from event logs generated by the Windows Event Log technology such as the Setup log mentioned in the above examples.

```
Get-EventLog -LogName System | Where-Object {$_['].EventID -eq 10148}
```

#### Related Cmdlets

[Clear-EventLog](#)

[Get-EventLog](#)

[Limit-EventLog](#)

[New-EventLog](#)

[Remove-EventLog](#)

[Show-EventLog](#)

[Write-EventLog](#)

## #70 Import-Module

### Import-Module

#### What can I do with it?

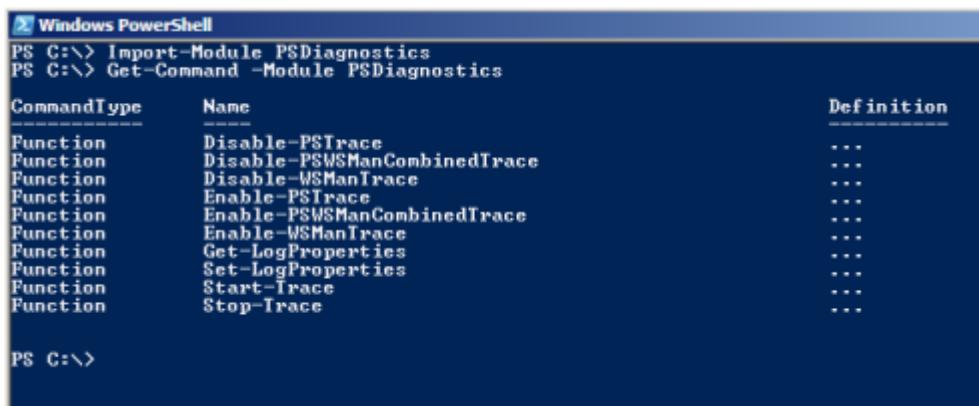
PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. Import-Module enables you to add one or more modules to your current session.

#### Examples:

Import the **PSDiagnostics** module and examine the newly available commands in the session from that module by using [Get-Module](#).

```
Import-Module PSDiagnostics
Get-Command -Module PSDiagnostics
```

You will see there are ten new functions available from that module.

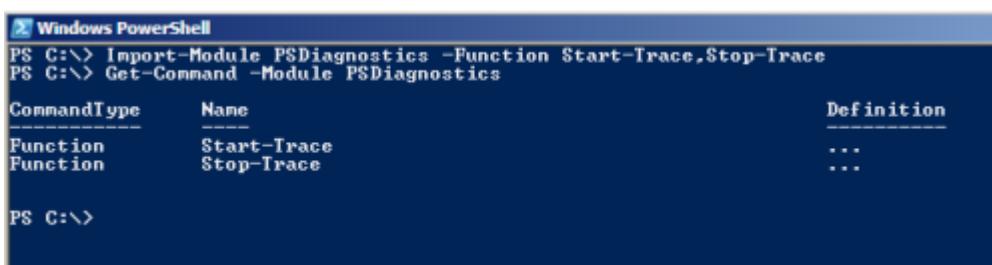


CommandType	Name	Definition
Function	Disable-PSTrace	...
Function	Disable-PSWSManCombinedTrace	...
Function	Disable-WSManTrace	...
Function	Enable-PSTrace	...
Function	Enable-PSWSManCombinedTrace	...
Function	Enable-WSManTrace	...
Function	Get-LogProperties	...
Function	Set-LogProperties	...
Function	Start-Trace	...
Function	Stop-Trace	...

Import only two functions, **Start-Trace** and **Stop-Trace** from the **PSDiagnostics** module and examine the newly available commands in the session from that module by using [Get-Module](#).

```
Import-Module PSDiagnostics -Function Start-Trace,Stop-Trace
Get-Command -Module PSDiagnostics
```

You will see that this time only those two functions are available.



CommandType	Name	Definition
Function	Start-Trace	...
Function	Stop-Trace	...

#### How could I have done this in PowerShell 1.0?

You could have used the Add-PSSnapin cmdlet to import custom snap-ins typically produced by third-parties. For example to import the popular Quest AD cmdlets snap-in you would use the below:

```
Add-PSSnapin Quest.ActiveRoles.ADManagement
```

## Related Cmdlets

[Get-Module](#)

[New-Module](#)

[Remove-Module](#)

[Export-ModuleMember](#)

## #71 New-Module

### New-Module

#### What can I do with it?

PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. New-Module enables you to create a dynamic module from a script block that is available in the current session.

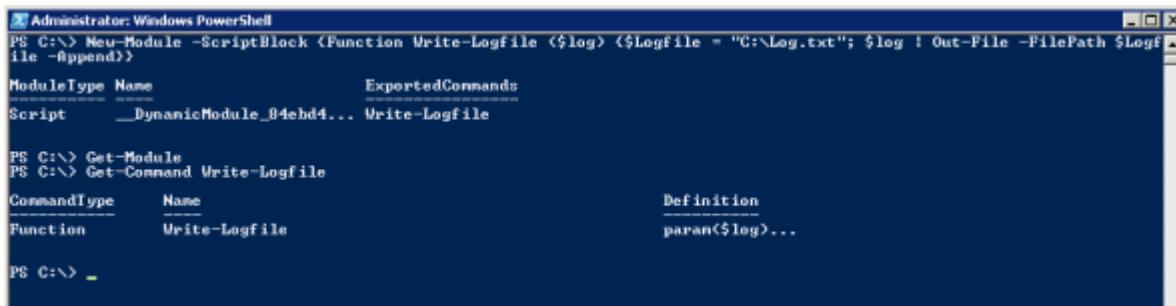
**Note:** New-Module does not create a module on disk available for use at a later date! However, [Jeffrey Snover](#) has created a [module](#) which will create a template for a new module on disk for you.

#### Examples:

Create a new dynamic module with the Function **Write-Logfile** as the scriptblock to create the module. Test to see whether the function is available from [Get-Module](#) or Get-Command.

```
New-Module -ScriptBlock
{Function Write-Logfile ($log) {$LogFile = "C:\Log.txt"; $log
| Out-File -FilePath $LogFile -Append}}
Get-Module
Get-Command Write-Logfile
```

You will see that [Get-Module](#) is not aware of the new module, but Get-Command is aware of the **Write-Logfile** function.



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is:

```
PS C:\> New-Module -ScriptBlock {Function Write-Logfile ($log) {$LogFile = "C:\Log.txt"; $log
| Out-File -FilePath $LogFile -Append}} -Name LogfileModule
PS C:\> Get-Module
PS C:\> Get-Command Write-Logfile
```

Output:

CommandType	Name	Definition
Function	Write-Logfile	param(\$log)...

Create a new dynamic module with the Function **Write-Logfile** as the scriptblock to create the module. Give it a name and use [Import-Module](#) to make it available to [Get-Module](#). Test to see whether the function is available from [Get-Module](#) or Get-Command.

```
New-Module -ScriptBlock
{Function Write-Logfile ($log) {$LogFile = "C:\Log.txt"; $log
| Out-File -FilePath $LogFile -Append}} -Name LogfileModule
| Import-Module
Get-Module
Get-Command Write-Logfile
```

You will see that this time both [Get-Module](#) and Get-Command are aware of the **LogfileModule** and **Write-Logfile** function.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> New-Module -ScriptBlock <Function Write-LogFile { \$log } > {\$Logfile = "C:\Log.txt"; \$log | Out-File -FilePath \$LogFile -Append} >> -Name LogfileModule | Import-Module is run. This creates a script-based module named LogfileModule. A table follows showing the module's properties:

ModuleType	Name	ExportedCommands
Script	LogFileModule	Write-LogFile

Another table shows the commands available in the module:

CommandType	Name	Definition
Function	Write-LogFile	param(\$log)...

## How could I have done this in PowerShell 1.0?

You could have created a custom snap-in and imported with the Add-PSSnapin cmdlet.

## Related Cmdlets

[Get-Module](#)

[Import-Module](#)

[Remove-Module](#)

[Export-ModuleMember](#)

## #72 Export-ModuleMember

### Export-ModuleMember

#### What can I do with it?

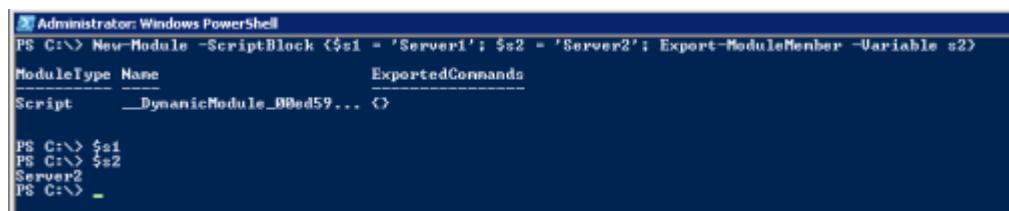
PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. Export-ModuleMember specifies elements from a module, like functions or variables, which can be exported. **Note:** This cmdlet can only be used within a \*.psm1 script module file or a dynamic module created with [New-Module](#).

#### Examples:

Create a new dynamic module using [New-Module](#) containing two variables inside the scriptblock. Export only the variable \$s2, so that it is available for use. **Note:** Export-ModuleMember needs to be included inside the scriptblock.

```
New-Module -ScriptBlock {$s1 = 'Server1'; $s2 = 'Server2';
Export-ModuleMember -Variable s2}
```

You will notice that \$s1 is not available in the current session, but \$s2 is.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". It displays the command "New-Module -ScriptBlock {\$s1 = 'Server1'; \$s2 = 'Server2'; Export-ModuleMember -Variable s2}" and its output. The output shows a table with one row. The table has two columns: "ModuleType" and "Name". The "ModuleType" column contains "Script" and the "Name" column contains "DynamicModule\_00ed59...". Below the table, the command "Export-ModuleMember -Variable s2" is shown again. At the bottom of the window, the variables \$s1 and \$s2 are listed with their values "Server1" and "Server2" respectively.

The other area to use this cmdlet is within a \*.psm1 script module file. In the below example by default all functions would be exported if the Export-ModuleMember cmdlet was not used. However, by using the Export-ModuleMember cmdlet we can control which functions are exported and also export aliases.

So, in the example below the **Write-Logfile** and **Greet-User** functions would be exported, but the **Yesterdays-Date** function would not. In addition the **gu** alias would be exported.

```
Function Write-Logfile ($log) {
    $Logfile = "C:\Log.txt"; $log | Out-File -FilePath $LogFile -Append

    Export-ModuleMember -Function Write-Logfile

Function Yesterdays-Date {
    (Get-Date).AddDays(-1)

Function Greet-User ($user) {
    Write-Host "Welcome" $user

    Set-Alias gu Greet-User
    Export-ModuleMember -Function Greet-User -Alias gu
```

## **How could I have done this in PowerShell 1.0?**

This functionality was not available with snap-ins in PowerShell 1.0

### **Related Cmdlets**

[Import-Module](#)

[Get-Module](#)

[Remove-Module](#)

## #73 New-ModuleManifest

### New-ModuleManifest

#### What can I do with it?

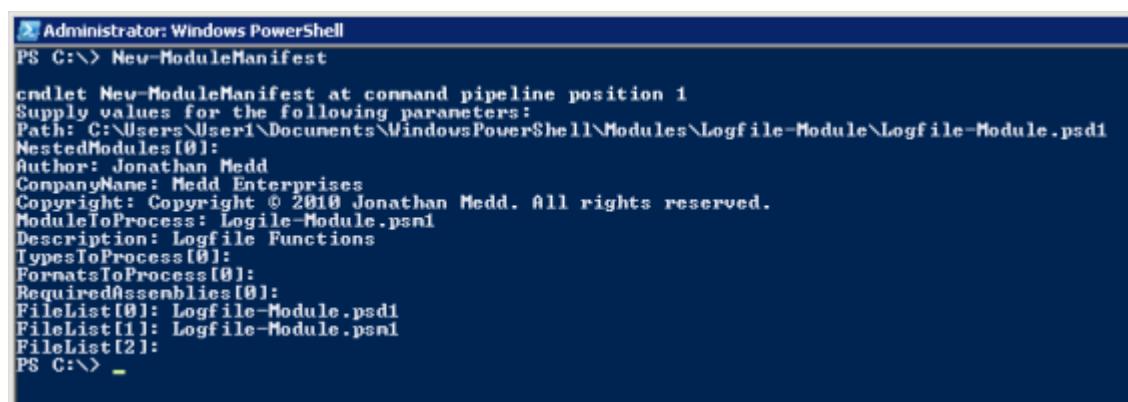
PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. Creators of modules can use the New-ModuleManifest cmdlet to create a module manifest \*.psd1 file. A module manifest file can be used to specify module configuration when the module is loaded. **Note:** More info about writing a module manifest can be found [here](#).

#### Example:

Create a new module manifest file using a mixture of default and specified values.

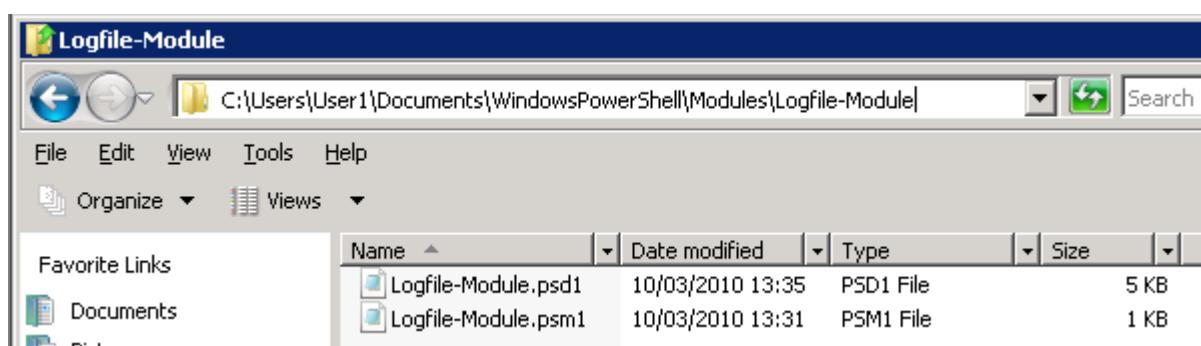
#### `New-ModuleManifest`

You will see that you are prompted to enter values to put in the manifest file and those which have defaults.



```
Administrator: Windows PowerShell
PS C:\> New-ModuleManifest
cmdlet New-ModuleManifest at command pipeline position 1
Supply values for the following parameters:
Path: C:\Users\User1\Documents\WindowsPowerShell\Modules\Logfile-Module\Logfile-Module.psd1
NestedModules[0]:
Author: Jonathan Medd
CompanyName: Medd Enterprises
Copyright: Copyright © 2010 Jonathan Medd. All rights reserved.
ModuleToProcess: Logfile-Module.psm1
Description: Logfile Functions
TypesToProcess[0]:
FormatsToProcess[0]:
RequiredAssemblies[0]:
FileList[0]: Logfile-Module.psd1
FileList[1]: Logfile-Module.psm1
FileList[2]:
PS C:\>
```

The \*.psd1 file is created in the standard module location  
`C:\Users\Username\Documents\WindowsPowerShell\Modules\` alongside the \*.psm1 containing the module.



The contents of the \*.psd1 file will look like the below:

```

#
# Module manifest for module 'Logfile-Module'
#
# Generated by: Jonathan Medd
#
# Generated on: 10/03/2010
#
@{

# Script module or binary module file associated with this manifest
ModuleToProcess = 'LogFile-Module.psm1'

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = '876e3d17-66ac-40f6-9e10-09913679011a'

# Author of this module
Author = 'Jonathan Medd'

# Company or vendor of this module
CompanyName = 'Medd Enterprises'

# Copyright statement for this module
Copyright = 'Copyright © 2010 Jonathan Medd. All rights reserved.'

# Description of the functionality provided by this module
Description = 'LogFile Functions'

# Minimum version of the Windows PowerShell engine required by this module
PowerShellVersion = ''

# Name of the Windows PowerShell host required by this module
PowerShellHostName = ''

# Minimum version of the Windows PowerShell host required by this module
PowerShellHostVersion = ''

# Minimum version of the .NET Framework required by this module
DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module
CLRVersion = ''

# Processor architecture (None, X86, Amd64, IA64) required by this module
ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to
# importing this module
RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to
# importing this module
ScriptsToProcess = @()

```

```

# Type files (.ps1xml) to be loaded when importing this module
TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
FormatsToProcess = @()

# Modules to import as nested modules of the module specified in
ModuleToProcess
NestedModules = @()

# Functions to export from this module
FunctionsToExport = '*'

# Cmdlets to export from this module
CmdletsToExport = '*'

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module
AliasesToExport = '*'

# List of all modules packaged with this module
ModuleList = @()

# List of all files packaged with this module
FileList = 'Logfile-Module.psd1', 'Logfile-Module.psm1'

# Private data to pass to the module specified in ModuleToProcess
PrivateData = ''

}


```

## **How could I have done this in PowerShell 1.0?**

This functionality was not available with snap-ins in PowerShell 1.0

### **Related Cmdlets**

[Import-Module](#)

[Get-Module](#)

[New-Module](#)

[Remove-Module](#)

[Export-ModuleMember](#)

[Test-ModuleManifest](#)

## #74 Test-ModuleManifest

### Test-ModuleManifest

#### What can I do with it?

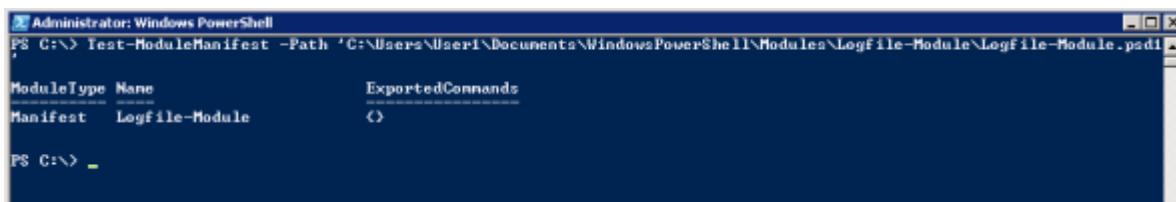
PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. A module creator could use `Test-ModuleManifest` to ensure that files listed in a \*.psd1 file, possibly created by [New-ModuleManifest](#), are valid.

#### Example:

Test that the **C:\Users\User1\Documents\WindowsPowerShell\Modules\Logfile-Module\Logfile-Module.psd1** (created in the [New-ModuleManifest](#) example) is valid.

```
Test-ModuleManifest -Path  
'C:\Users\User1\Documents\WindowsPowerShell\Modules\Logfile-Module\Logfile-  
Module.psd1'
```

You will see that this returns an object for the module. If any files were not valid then an error message would be produced.

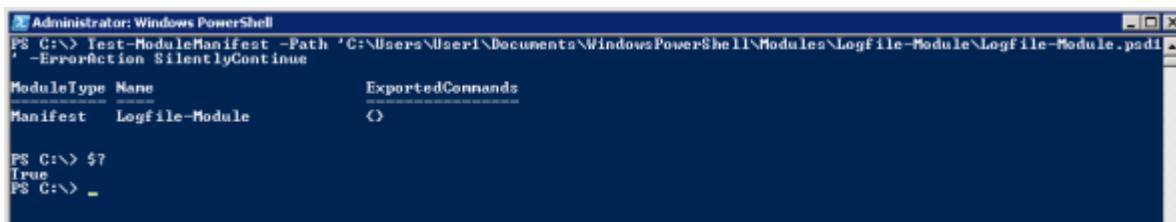


ModuleType	Name	ExportedCommands
Manifest	LogFile-Module	<-->

To obtain a tidy **True** or **False** answer the `ErrorAction` parameter can be used with the **SilentlyContinue** value to suppress any errors. Then examine the current value of `$?` automatic variable which contains the execution status of the last operation. It contains **True** if the last operation succeeded.

```
Test-ModuleManifest -Path  
'C:\Users\User1\Documents\WindowsPowerShell\Modules\Logfile-Module\Logfile-  
Module.psd1'  
-ErrorAction SilentlyContinue  
$?
```

You will see the result in this case is **True**.



ModuleType	Name	ExportedCommands
Manifest	LogFile-Module	<-->

PS C:\> \$?  
True  
PS C:\>

#### How could I have done this in PowerShell 1.0?

This functionality was not available with snap-ins in PowerShell 1.0

## **Related Cmdlets**

[Import-Module](#)

[Get-Module](#)

[New-Module](#)

[Remove-Module](#)

[Export-ModuleMember](#)

[New-ModuleManifest](#)

## #75 Remove-Module

### [Remove-Module](#)

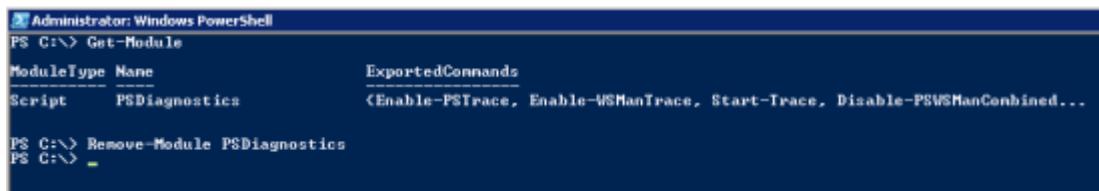
#### What can I do with it?

PowerShell 2.0 introduces the concept of modules; essentially they are the evolution of snap-ins from PowerShell 1.0. Remove-Module enables you to remove a module, and all of its functions, variables etc, previously imported with [Import-Module](#).

#### Example:

Check currently available modules with [Get-Module](#) and remove the **PSDiagnostics** module.

```
Get-Module  
Remove-Module PSDiagnostics
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Module" is run, displaying a table with columns "ModuleType", "Name", and "ExportedCommands". One row shows "Script" under "ModuleType", "PSDiagnostics" under "Name", and "(Enable-PSTrace, Enable-WSMTrace, Start-Trace, Disable-PSWSManCombined...)" under "ExportedCommands". The command "Remove-Module PSDiagnostics" is then run, and the prompt "PS C:\>" appears again.

#### How could I have done this in PowerShell 1.0?

If you had imported a PSSnapin with Add-PSSnapin you could remove it with Remove-PSSnapin.

#### Related Cmdlets

[Get-Module](#)

[Import-Module](#)

## #76 Stop-Computer

[Stop-Computer](#)

### What can I do with it?

Shutdown a local or remote computer

#### Example:

Immediately shut down the computer Server01.

```
Stop-Computer -ComputerName Server01 -Force
```

### How could I have done this in PowerShell 1.0?

You could have used the **Win32\_OperatingSystem** [WMI Class](#) and the **Win32Shutdown** method.

```
(Get-WmiObject -Class Win32_OperatingSystem  
-ComputerName Server01).Win32Shutdown(5)
```

Alternatively the Sysinternals tool [PSShutdown](#) could be used to shut down a local or remote computer.

### Related Cmdlets

[Add-Computer](#)

[Checkpoint-Computer](#)

[Remove-Computer](#)

[Restart-Computer](#)

[Restore-Computer](#)

[Test-Connection](#)

## #77 Remove-Computer

[Remove-Computer](#)

### What can I do with it?

Remove the local computer from a workgroup or domain.

#### Example:

Remove the local computer from the current domain, and then reboot to make the change take effect using the [Restart-Computer](#) cmdlet.

`Remove-Computer; Restart-Computer`

### How could I have done this in PowerShell 1.0?

You could have used the **Win32\_ComputerSystem** [WMI Class](#) and the **UnjoinDomainOrWorkgroup** method. **Note:** Make sure you run PowerShell with elevated privileges otherwise it will be unsuccessful and you will receive a return value of 5 rather than 0 for a success.

```
(Get-WmiObject -Class Win32_ComputerSystem).UnjoinDomainOrWorkgroup($null,$null,0)
```

Alternatively you could use the command line tool netdom to remove a computer from a domain:

```
NETDOM remove /userd:adminuser /passwordd:apassword
```

### Related Cmdlets

[Add-Computer](#)

[Checkpoint-Computer](#)

[Restart-Computer](#)

[Restore-Computer](#)

[Stop-Computer](#)

[Test-Connection](#)

## #78 Start-Transaction

### Start-Transaction

#### What can I do with it?

PowerShell 2.0 introduces new functionality in the form of transactions. By grouping together a set of commands to form a transaction they can either all be committed or all rolled back depending on success. Both cmdlets and providers can support transactions; cmdlets will have the **UseTransaction** parameter. To identify which cmdlets support transactions run the following:

```
Get-Command | Where-Object {$_ .Definition -match 'UseTransaction'}
```

And for providers:

```
Get-PSPrinter | Where-Object {$_ .Capabilities -like '*transactions*'}
```

Start-Transaction begins a transaction.

#### Example:

A good example of a possible use for transactions is within the registry (In fact it is the only provider as of the release of Windows Server 2008 R2 which has transactions enabled). Change directory into the registry provider. Begin a new transaction and use the New-Item and New-ItemProperty cmdlets to potentially create entries within the registry.

```
cd HKLM:\Software
Start-Transaction
New-Item Test -UseTransaction
New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> cd HKLM:\Software
PS HKLM:\Software> Start-Transaction
Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS HKLM:\Software> New-Item Test -UseTransaction

Hive: HKEY_LOCAL_MACHINE\Software

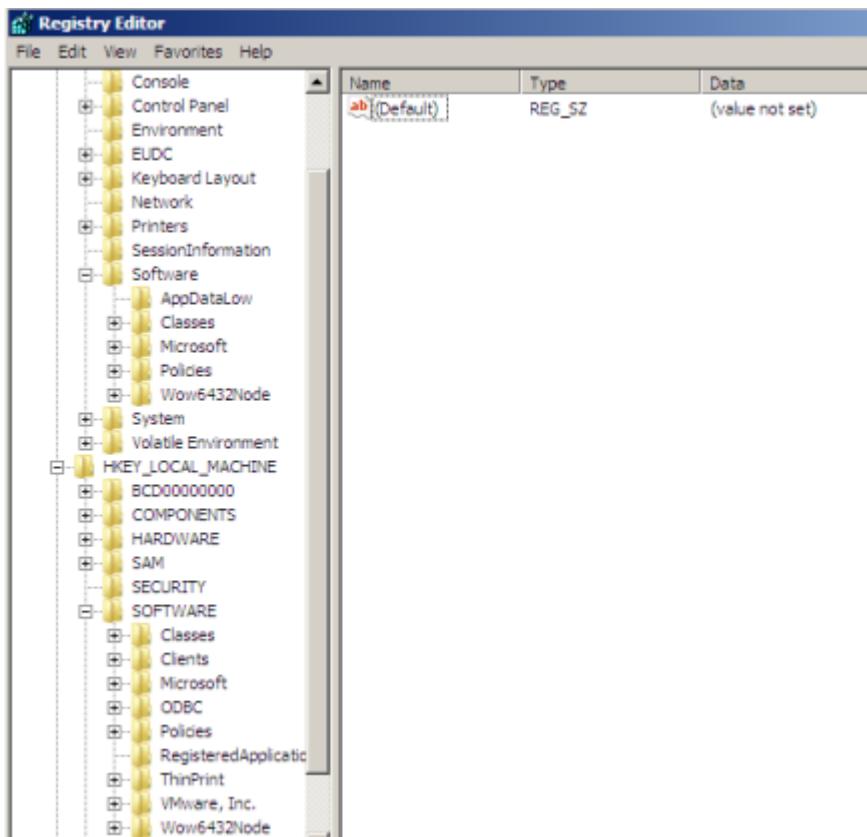
  SMC  UC Name          Property
  --  --  --
  0    0  Test           0

PS HKLM:\Software> New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
PSCChildName : Test
PSPDrive    : HKLM
PSPProvider  : Microsoft.PowerShell.Core\Registry
TestKey     : 1000

PS HKLM:\Software>
```

You will notice that since we have not yet completed the transaction no changes have yet been made in the registry.



## How could I have done this in PowerShell 1.0?

Transactional functionality was not available in PowerShell 1.0.

### Related Cmdlets

[Get-Transaction](#)

[Complete-Transaction](#)

[Undo-Transaction](#)

[Use-Transaction](#)

## #79 Complete-Transaction

### Complete-Transaction

#### What can I do with it?

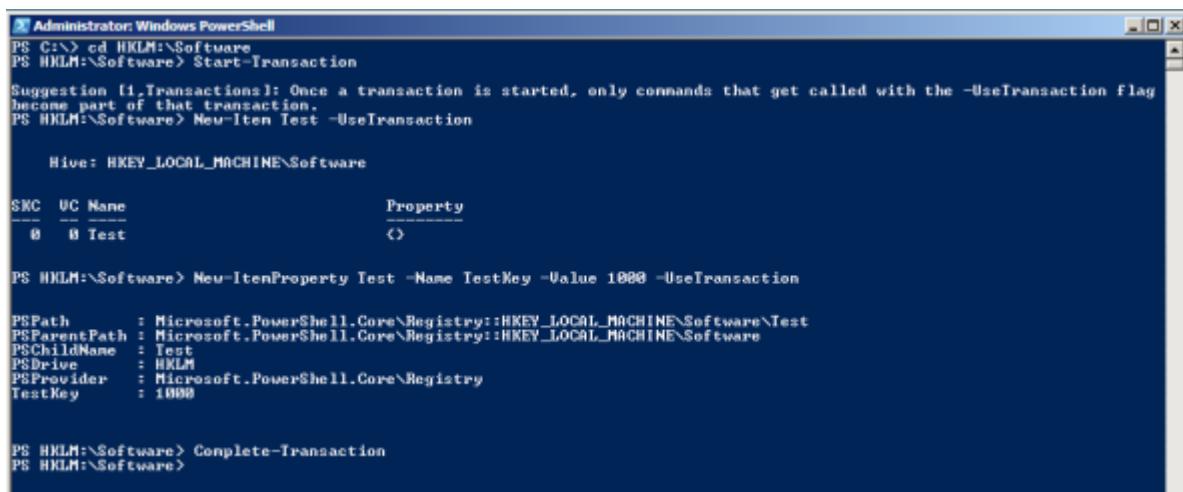
PowerShell 2.0 introduces new functionality in the form of transactions. By grouping together a set of commands to form a transaction they can either all be committed or all rolled back depending on success.

Complete-Transaction commits a transaction which has been kicked off with [Start-Transaction](#).

#### Example:

A good example of a possible use for transactions is within the registry. Change directory into the registry provider. Begin a new transaction and use the New-Item and New-ItemProperty cmdlets to potentially create entries within the registry. Use Complete-Transaction to commit these changes.

```
cd HKLM:\Software
Start-Transaction
New-Item Test -UseTransaction
New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction
Complete-Transaction
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> cd HKLM:\Software
PS HKLM:\Software> Start-Transaction
Suggestion [1]: Transactions: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS HKLM:\Software> New-Item Test -UseTransaction

    Hive: HKEY_LOCAL_MACHINE\Software

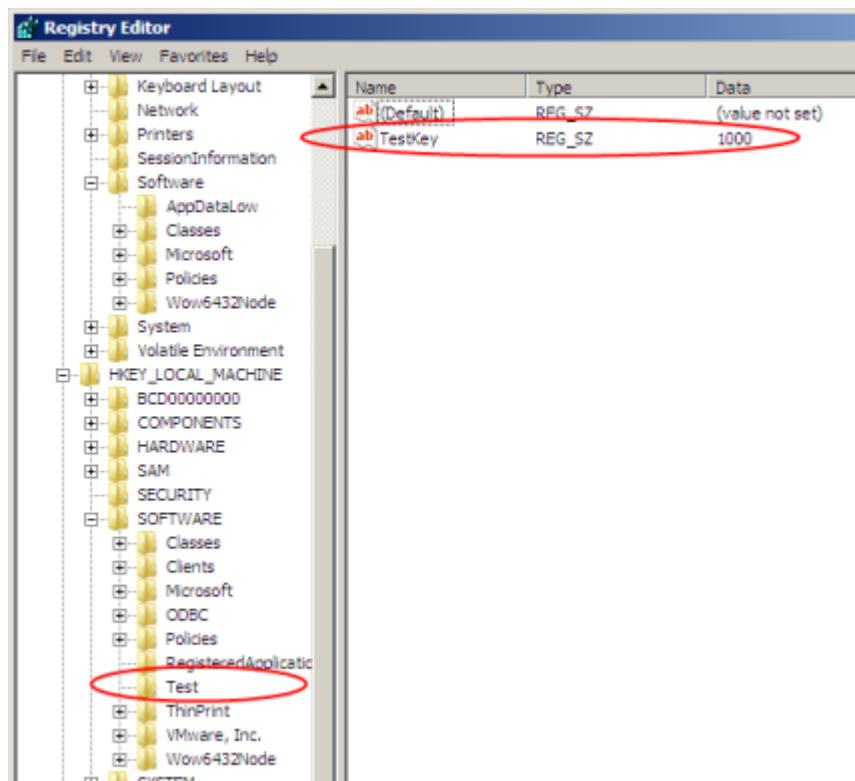
SKC UC Name          Property
--- -- -----
  0   0 Test

PS HKLM:\Software> New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
PSChildName : Test
PSDrive     : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
TestKey     : 1000

PS HKLM:\Software> Complete-Transaction
PS HKLM:\Software>
```

You will notice that after completing the transaction the changes have been made in the registry.



## How could I have done this in PowerShell 1.0?

Transactional functionality was not available in PowerShell 1.0.

### Related Cmdlets

[Get-Transaction](#)

[Start-Transaction](#)

[Undo-Transaction](#)

[Use-Transaction](#)

## #80 Get-Transaction

### [Get-Transaction](#)

#### What can I do with it?

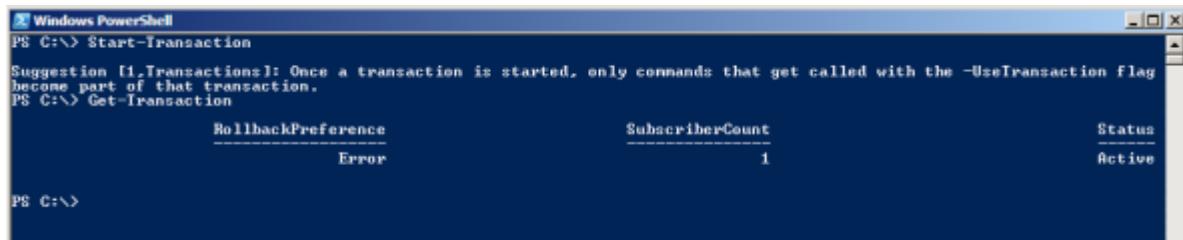
PowerShell 2.0 introduces new functionality in the form of transactions. By grouping together a set of commands to form a transaction they can either all be committed or all rolled back depending on success.

Get-Transaction returns an object of a current transaction which has been kicked off with [Start-Transaction](#).

#### Examples:

Start a transaction then use Get-Transaction to examine its details.

```
Start-Transaction  
Get-Transaction
```



RollbackPreference	SubscriberCount	Status
Error	1	Active

Another example of a possible use for transactions is within the registry. Change directory into the registry provider. Begin a new transaction and use the New-Item and New-ItemProperty cmdlets to potentially create entries within the registry. Use [Complete-Transaction](#) to commit these changes. Use Get-Transaction to retrieve the details, notice that the status is **Committed**.

```
cd HKLM:\Software  
Start-Transaction  
New-Item Test -UseTransaction  
New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction  
Complete-Transaction  
Get-Transaction
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> cd HKLM:\Software
PS HKLM:\Software> Start-Transaction
Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS HKLM:\Software> New-Item Test -UseTransaction

    Hive: HKEY_LOCAL_MACHINE\Software

SNC UC Name          Property
--- -- -----
  0  0 Test           <>

PS HKLM:\Software> New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
PSChildName  : Test
PSDrive     : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
TestKey     : 1000

PS HKLM:\Software> Complete-Transaction
PS HKLM:\Software> Get-Transaction

    RollbackPreference   SubscriberCount   Status
    -----              -----             -----
          Error                   0            Committed
```

A red circle highlights the "Status" column, specifically the "Committed" entry.

## How could I have done this in PowerShell 1.0?

Transactional functionality was not available in PowerShell 1.0.

### Related Cmdlets

[Complete-Transaction](#)

[Start-Transaction](#)

[Undo-Transaction](#)

[Use-Transaction](#)

## #81 Undo-Transaction

### Undo-Transaction

#### What can I do with it?

PowerShell 2.0 introduces new functionality in the form of transactions. By grouping together a set of commands to form a transaction they can either all be committed or all rolled back depending on success.

Undo-Transaction rolls back the active transaction.

#### Example:

A good example of a possible use for transactions is within the registry. Change directory into the registry provider. Begin a new transaction and use the New-Item and New-ItemProperty cmdlets to potentially create entries within the registry. Use [Get-Transaction](#) to view details of the current transaction.

```
cd HKLM:\Software
Start-Transaction
New-Item Test -UseTransaction
New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction
Get-Transaction
```

You will notice that there is currently **1** subscriber and the status is **Active**.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command history is as follows:

```
PS C:\> cd HKLM:\Software
PS HKLM:\Software> Start-Transaction
Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS HKLM:\Software> New-Item Test -UseTransaction
Hive: HKEY_LOCAL_MACHINE\Software

SNC UC Name          Property
  0  0 Test           O

PS HKLM:\Software> New-ItemProperty Test -Name TestKey -Value 1000 -UseTransaction

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
PSChildName : Test
PSDrive     : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
TestKey     : 1000

PS HKLM:\Software> Get-Transaction
RollbackPreference          SubscribersCount          Status
Error                           1                         Active
```

Start a new transaction, use the New-ItemProperty cmdlets to potentially create another new entry within the registry and use [Get-Transaction](#) to view details of the current transaction.

```
Start-Transaction
New-ItemProperty Test -Name TestKey2 -Value 2000 -UseTransaction
Get-Transaction
```

You will notice that there are now **2** subscribers and the status is still **Active**.

```

Administrator: Windows PowerShell
PS HKLM:\Software> Start-Transaction
Suggestion [i,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS HKLM:\Software> New-ItemProperty Test -Name TestKey2 -Value 2000 -UseTransaction

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
PSChildName  : Test
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
TestKey2     : 2000

PS HKLM:\Software> Get-Transaction
    RollbackPreference          SubscriberCount          Status
                           Error                               2           Active

```

Now use Undo-Transaction to roll back the changes and [Get-Transaction](#) to view details of the current transaction

[Undo-Transaction](#)  
[Get-Transaction](#)

Notice that it has rolled back the changes for both transactions and the status is now **RolledBack**.

```

Administrator: Windows PowerShell
PS HKLM:\Software> Undo-Transaction
PS HKLM:\Software> Get-Transaction
    RollbackPreference          SubscriberCount          Status
                           Error                               0           RolledBack

```

## How could I have done this in PowerShell 1.0?

Transactional functionality was not available in PowerShell 1.0.

## Related Cmdlets

[Get-Transaction](#)

[Complete-Transaction](#)

[Start-Transaction](#)

[Use-Transaction](#)

## #82 Use-Transaction

### Use-Transaction

#### What can I do with it?

PowerShell 2.0 introduces new functionality in the form of transactions. By grouping together a set of commands to form a transaction they can either all be committed or all rolled back depending on success.

Use-Transaction enables you to add a scriptblock to a transaction. **Note:** This only works with transaction-enabled .NET Framework objects such as [Microsoft.PowerShell.Commands.Management.TransactedString](#).

You will see below the difference between a transacted string object and a normal string object, i.e. there are a few options to manipulate it with.

The screenshot shows a Windows PowerShell window with two sections of output from the Get-Member cmdlet. The first section is for the type `Microsoft.PowerShell.Commands.Management.TransactedString`, and the second is for the type `System.String`. Both sections show a table with columns for Name, MemberType, and Definition.

Name	MemberType	Definition
Append	Method	System.Void Append(string text)
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
Remove	Method	System.Void Remove(int startIndex, int length)
ToString	Method	string ToString()
Length	Property	System.Int32 Length {get;}

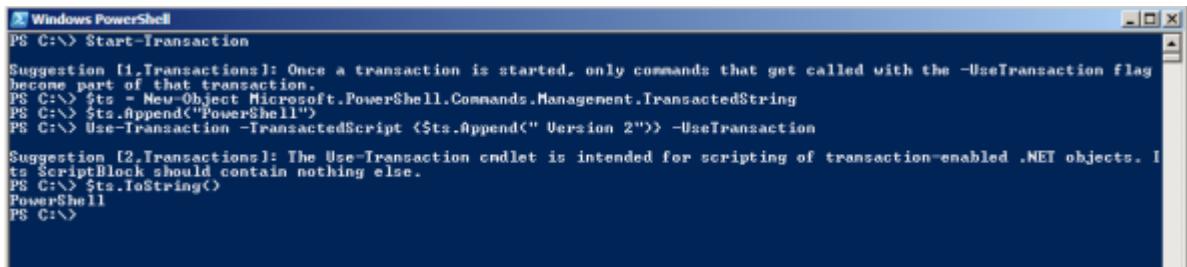
Name	MemberType	Definition
Clone	Method	System.Object Clone()
CompareTo	Method	int CompareTo(Systen.Object value), int CompareTo(string strB)
Contains	Method	bool Contains(string value)
CopyTo	Method	System.Void CopyTo(int sourceIndex, char[] destination, int destinationIndex,...)
EndsWith	Method	bool EndsWith(string value), bool EndsWith(string value, System.StringCompari...)
Equals	Method	bool Equals(System.Object obj), bool Equals(string value), bool Equals(string...)
GetEnumerator	Method	System.CharEnumerator GetEnumerator()
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode()
IndexOf	Method	int IndexOf(char value), int IndexOf(char value, int startIndex), int IndexOf...
IndexOfAny	Method	int IndexOfAny(char[] anyOf), int IndexOfAny(char[] anyOf, int startIndex), i...
Insert	Method	string Insert(int startIndex, string value)
IsNormalized	Method	bool IsNormalized(), bool IsNormalized(System.Text.NormalizationForm normaliz...)
LastIndexOf	Method	int LastIndexOf(char value), int LastIndexOf(char value, int startIndex), int...
LastIndexOfAny	Method	int LastIndexOfAny(char[] anyOf), int LastIndexOfAny(char[] anyOf, int startI...
Normalize	Method	string Normalize(), string Normalize(System.Text.NormalizationForm normaliz...
PadLeft	Method	string PadLeft(int totalWidth), string PadLeft(int totalWidth, char paddingChar)
PadRight	Method	string PadRight(int totalWidth), string PadRight(int totalWidth, char padding...
Remove	Method	string Remove(int startIndex, int count), string Remove(int startIndex)
Replace	Method	string Replace(char oldChar, char newChar), string Replace(string oldValue, z...
Split	Method	string[] Split(params char[] separator), string[] Split(char[] separator, int...)
StartsWith	Method	bool StartsWith(string value), bool StartsWith(string value, System.StringCom...)
Substring	Method	string Substring(int startIndex), string Substring(int startIndex, int length)
ToCharArray	Method	char[] ToCharArray(), char[] ToCharArray(int startIndex, int length)
ToLower	Method	string ToLower(), string ToLower(System.Globalization.CultureInfo culture)

#### Example:

Start a transaction and create a new transacted string. Add the text 'PowerShell' to the string, and then add the text ' Version 2' with the Use-Transaction cmdlet.

```
Start-Transaction
$sts = New-Object Microsoft.PowerShell.Commands.Management.TransactedString
$sts.Append("PowerShell")
Use-Transaction -TransactedScript {$sts.Append(" Version 2")} -
UseTransaction
$sts.ToString()
```

Note that the current value of the string only contains the text 'PowerShell'.

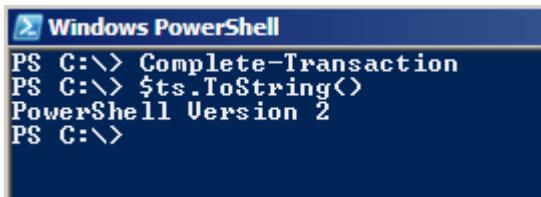


```
PS C:\> Start-Transaction
Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
PS C:\> $ts = New-Object Microsoft.PowerShell.Commands.Management.TransactedString
PS C:\> $ts.AppendLine("PowerShell")
PS C:\> Use-Transaction -TransactedScript $ts.AppendLine(" Version 2") -UseTransaction
Suggestion [2,Transactions]: The Use-Transaction cmdlet is intended for scripting of transaction-enabled .NET objects. I
ts ScriptBlock should contain nothing else.
PS C:\> $ts.ToString()
PowerShell
PS C:\>
```

Now complete the transaction and again view the current value of the string.

```
Complete-Transaction
$ts.ToString()
```

You will notice that the value of the string now contains all the text 'PowerShell Version 2'.



```
PS C:\> Complete-Transaction
PS C:\> $ts.ToString()
PowerShell Version 2
PS C:\>
```

## How could I have done this in PowerShell 1.0?

Transactional functionality was not available in PowerShell 1.0.

## Related Cmdlets

[Get-Transaction](#)

[Complete-Transaction](#)

[Start-Transaction](#)

[Undo-Transaction](#)

## #83 ConvertTo-CSV

### ConvertTo-CSV

#### What can I do with it?

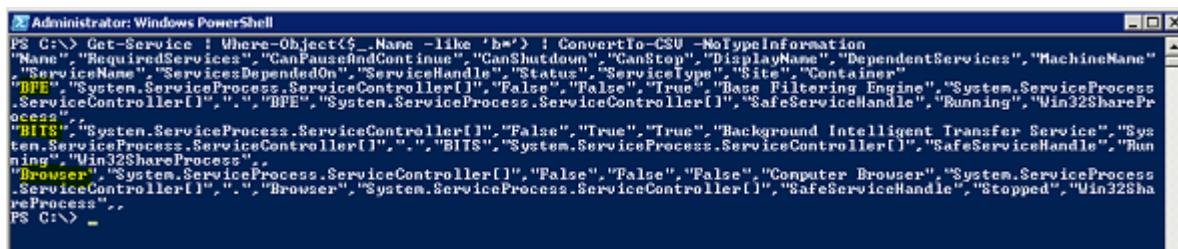
Convert a .NET object into a series of CSV style strings, stored in memory.

#### Example:

Retrieve a list of services beginning with the letter **b** and convert the object into CSV style strings

```
Get-Service | Where-Object {$_.Name -like 'b*' }  
| ConvertTo-CSV -NoTypeInformation
```

You will notice that the data returned from the services has been converted into strings separated by a comma. (The names of the services are highlighted in yellow.)



```
Administrator: Windows PowerShell  
PS C:\> Get-Service | Where-Object {$_.Name -like 'b*' } | ConvertTo-CSV -NoTypeInformation  
"Name","RequiredServices","CanPauseAndContinue","CanShutdown","CanStop","DisplayName","DependentServices","MachineName"  
"ServiceName","ServicesDependedOn","ServiceHandle","Status","ServiceType","Site","Container"  
"BFE","System.ServiceProcess.ServiceController[1]","False","False","True","Base Filtering Engine","System.ServiceProcess.  
ServiceController[1]",".","BFE","System.ServiceProcess.ServiceController[1]","SafeServiceHandle","Running","Win32SharePr  
ocess","BITS","System.ServiceProcess.ServiceController[1]","False","True","Background Intelligent Transfer Service","Sys  
tem.ServiceProcess.ServiceController[1]",".","BITS","System.ServiceProcess.ServiceController[1]","SafeServiceHandle","Run  
ning","Win32ShareProcess"  
"Browser","System.ServiceProcess.ServiceController[1]","False","False","False","Computer Browser","System.ServiceProcess.  
.ServiceController[1]",".","Browser","System.ServiceProcess.ServiceController[1]","SafeServiceHandle","Stopped","Win32Sha  
reProcess"
```

#### How could I have done this in PowerShell 1.0?

You could have used Export-CSV, but that would have put the information into a file.

#### Related Cmdlets

### Import-CSV

### Export-CSV

### ConvertFrom-CSV

## #84 ConvertFrom-CSV

### [ConvertFrom-CSV](#)

#### What can I do with it?

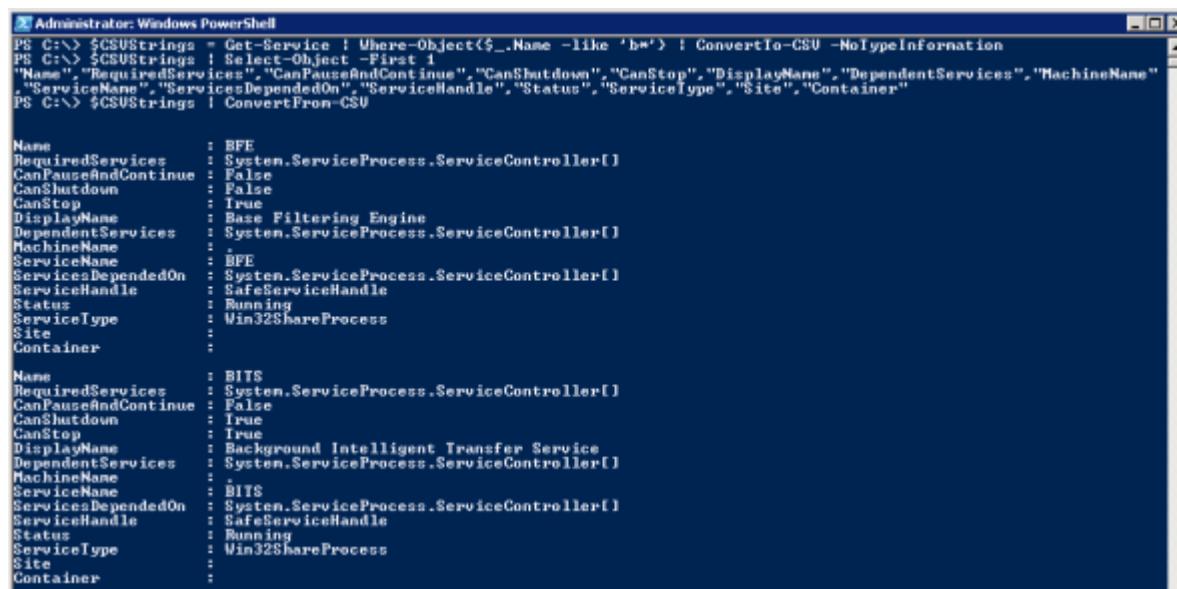
Convert a series of CSV style strings which have been generated by [ConvertTo-CSV](#) back into objects.

#### Example:

Retrieve a list of services beginning with the letter **b** and convert the object into CSV style strings, storing them into the variable **\$CSVStrings** . Convert these back into objects.

```
$CSVStrings = Get-Service | Where-Object {$__.Name -like 'b*' }  
| ConvertTo-CSV -NoTypeInformation  
$CSVStrings | Select-Object -First 1  
$CSVStrings | ConvertFrom-CSV
```

You will notice that **\$CSVStrings** contains the same data as for the example in [ConvertTo-CSV](#) , cut short for clarity. That variable is piped into [ConvertFrom-CSV](#) to change it back.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $CSVStrings = Get-Service | Where-Object {$__.Name -like 'b*' } | ConvertTo-CSV -NoTypeInformation  
PS C:\> $CSVStrings | Select-Object -First 1  
"Name","RequiredServices","CanPauseAndContinue","CanShutdown","CanStop","DisplayName","DependentServices","MachineName","ServiceName","ServicesDependedOn","ServiceHandle","Status","ServiceType","Site","Container"  
PS C:\> $CSVStrings | ConvertFrom-CSV
```

The output displays two sets of service properties as CSV objects. The first set is for the BFE service, and the second is for the BITS service. The properties listed are:

Property	BFE	BITS
Name	BFE	BITS
RequiredServices	System.ServiceProcess.ServiceController[]	System.ServiceProcess.ServiceController[]
CanPauseAndContinue	False	False
CanShutdown	False	True
CanStop	True	True
DisplayName	Base Filtering Engine	Background Intelligent Transfer Service
DependentServices	System.ServiceProcess.ServiceController[]	System.ServiceProcess.ServiceController[]
MachineName	.	.
ServiceName	BFE	BITS
ServicesDependedOn	System.ServiceProcess.ServiceController[]	System.ServiceProcess.ServiceController[]
ServiceHandle	SafeServiceHandle	SafeServiceHandle
Status	Running	Running
ServiceType	Win32ShareProcess	Win32ShareProcess
Site	.	.
Container	.	.

#### How could I have done this in PowerShell 1.0?

You could have used Import-CSV, but that would have read the information from a file.

#### Related Cmdlets

##### [Import-CSV](#)

##### [Export-CSV](#)

##### [ConvertTo-CSV](#)

## #85 ConvertFrom-StringData

### ConvertFrom-StringData

#### What can I do with it?

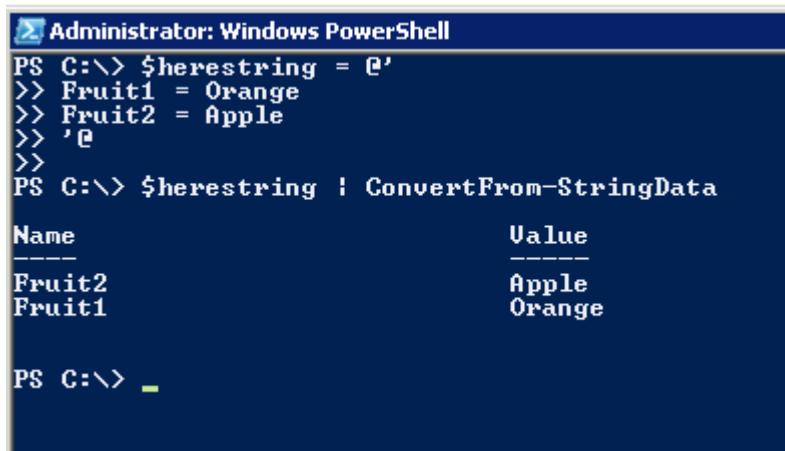
Converts a string which contains one or multiple key and value pairs into a hash table. Input is typically from a [here-string](#) since each key and value must be on a separate line.

#### Example:

Create a here-string and store it in the variable \$herestring. Convert it into a hash table.

```
$herestring = @'  
Fruit1 = Orange  
Fruit2 = Apple  
'@  
  
$herestring | ConvertFrom-StringData
```

You will notice that the data is now in the form of a hash table



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

```
PS C:\> $herestring = @'  
>> Fruit1 = Orange  
>> Fruit2 = Apple  
>> '@  
>>  
PS C:\> $herestring | ConvertFrom-StringData
```

The output is a table:

Name	Value
Fruit2	Apple
Fruit1	Orange

PS C:\> \_

#### How could I have done this in PowerShell 1.0?

To create a new hash table you could use a new .Net object of type [System.Collections.Hashtable](#), something like the below

```
$strings = ('Fruit1 = Orange', 'Fruit2 = Apple')  
$table = New-Object System.Collections.Hashtable  
  
foreach ($string in $strings) {  
  
    $split = $string.split("=")  
    $part1 = $split[0]  
    $part2 = $split[1]  
    $table.add("$part1", "$part2")  
}
```

## #86 ConvertTo-XML

### ConvertTo-XML

#### What can I do with it?

Convert a .NET object into an XML-based representation of it.

#### Example:

Retrieve a list of services beginning with the letter **b** and convert the object into an XML-based representation. Use the available **Save** method of the XML object to save the data into an XML file.

```
$xml = Get-Service | Where-Object {$_.Name -like 'b*'} | ConvertTo-Xml  
$xml.Save("C:\temp\service.xml")
```

You can see that when opened the file is a typical style XML document

```
<?xml version="1.0" ?>  
- <Objects>  
- <Object Type="System.ServiceProcess.ServiceController">  
  <Property Name="Name" Type="System.String">BFE</Property>  
  - <Property Name="RequiredServices" Type="System.ServiceProcess.ServiceController[]">  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
  </Property>  
  <Property Name="CanPauseAndContinue" Type="System.Boolean">False</Property>  
  <Property Name="CanShutdown" Type="System.Boolean">False</Property>  
  <Property Name="CanStop" Type="System.Boolean">True</Property>  
  <Property Name="DisplayName" Type="System.String">Base Filtering Engine</Property>  
  - <Property Name="DependentServices" Type="System.ServiceProcess.ServiceController[]">  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
  </Property>  
  <Property Name="MachineName" Type="System.String">.</Property>  
  <Property Name="ServiceName" Type="System.String">BFE</Property>  
  - <Property Name="ServicesDependedOn" Type="System.ServiceProcess.ServiceController[]">  
    <Property Type="System.ServiceProcess.ServiceController">System.ServiceProcess.ServiceController</Property>  
  </Property>  
  <Property Name="ServiceHandle" Type="SafeServiceHandle">SafeServiceHandle</Property>  
  <Property Name="Status" Type="System.ServiceProcess.ServiceControllerStatus">Running</Property>  
  <Property Name="ServiceType" Type="System.ServiceProcess.ServiceType">Win32ShareProcess</Property>  
  <Property Name="Site" Type="System.ComponentModel.ISite" />  
  <Property Name="Container" Type="System.ComponentModel.IContainer" />  
</Object>
```

#### How could I have done this in PowerShell 1.0?

You could have used Export-Clixml, but that would have exported the information directly to a file which typically did not have the correct formatting when the document was viewed. It was more intended to be used in conjunction with Import-Clixml to recreate the original object from a file.

## **Related Cmdlets**

[Export-Clixml](#)

[Import-Clixml](#)

[ConvertTo-Html](#)

[ConvertTo-Csv](#)

## #87 Get-FormatData

### [Get-FormatData](#)

#### What can I do with it?

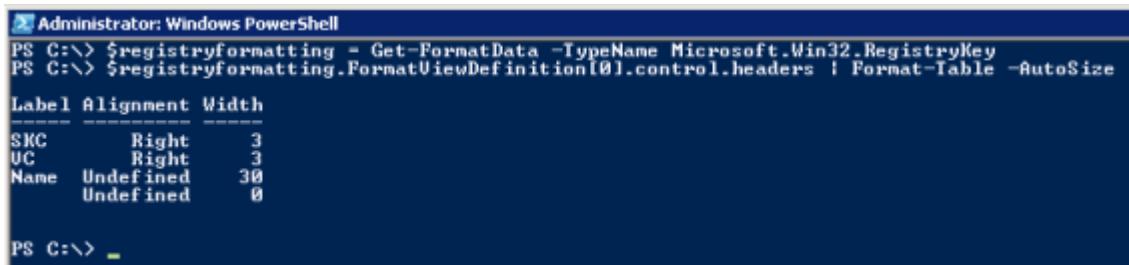
Retrieve format data from the current session. Within a session formatting data could include formatting from \*.ps1xml format files stored in the PowerShell installation directory, formatting from imported modules or snap-ins, or formatting from commands imported with [Import-PSSession](#).

#### Example:

Retrieve the formatting for the TypeName **Microsoft.Win32.RegistryKey** and view some of its properties.

```
$registryformatting = Get-FormatData -TypeName Microsoft.Win32.RegistryKey
$registryformatting.FormatViewDefinition[0].control.headers
| Format-Table -AutoSize
```

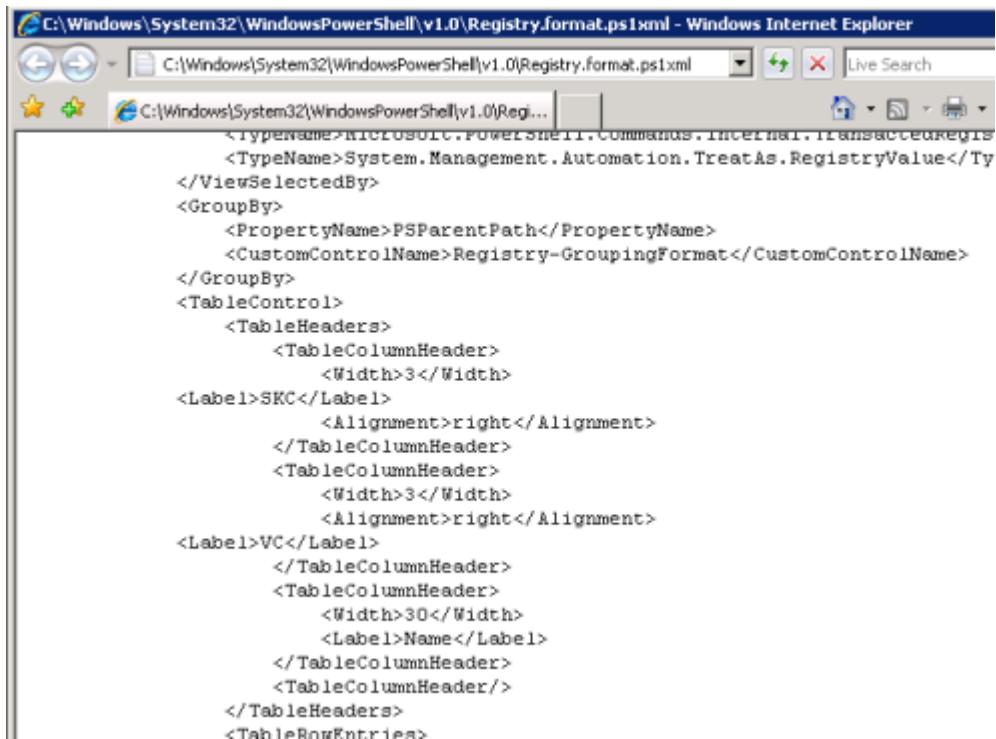
You will notice that some properties are obtained by drilling down into its XML style format.



Label	Alignment	Width
SKC	Right	3
UC	Right	3
Name	Undefined	30
	Undefined	0

This same data can be viewed in the file **Registry.format.ps1xml** which is located in the PowerShell installation folder. You can determine this location by examining the **\$pshome** variable, typically it is **C:\Windows\System32\WindowsPowerShell\v1.0** .

Notice below the same data in the file **Registry.format.ps1xml**.



The screenshot shows a Windows Internet Explorer window displaying the XML file C:\Windows\System32\WindowsPowerShell\v1.0\Registry.format.ps1xml. The XML code defines a PowerShell cmdlet for viewing registry keys. It includes sections for TypeName, ViewSelectedBy, GroupBy, TableControl, and TableHeaders, specifying widths and alignment for columns labeled SKC and VC.

```
< TypeName="Microsoft.PowerShell.Commands.Internal.TransactedRegis
< TypeName="System.Management.Automation.TreatAs.RegistryValue">
</ViewSelectedBy>
<GroupBy>
    <PropertyName>PSParentPath</PropertyName>
    <CustomControlName>Registry-GroupingFormat</CustomControlName>
</GroupBy>
<TableControl>
    <TableHeaders>
        <TableColumnHeader>
            <Width>3</Width>
        </TableColumnHeader>
        <TableColumnHeader>
            <Width>3</Width>
            <Alignment>right</Alignment>
        </TableColumnHeader>
        <TableColumnHeader>
            <Width>30</Width>
            <Label>Name</Label>
        </TableColumnHeader>
        <TableColumnHeader/>
    </TableHeaders>
    <TableRowEntries>
```

## How could I have done this in PowerShell 1.0?

In the example above you could have viewed the **Registry.format.ps1xml** in an XML viewer or used

```
Get-Content  
C:\Windows\System32\WindowsPowerShell\v1.0\Registry.format.ps1xml
```

to read in the XML file.

## Related Cmdlets

[Export-FormatData](#)

[Update-FormatData](#)

## #88 Export-FormatData

### [Export-FormatData](#)

#### What can I do with it?

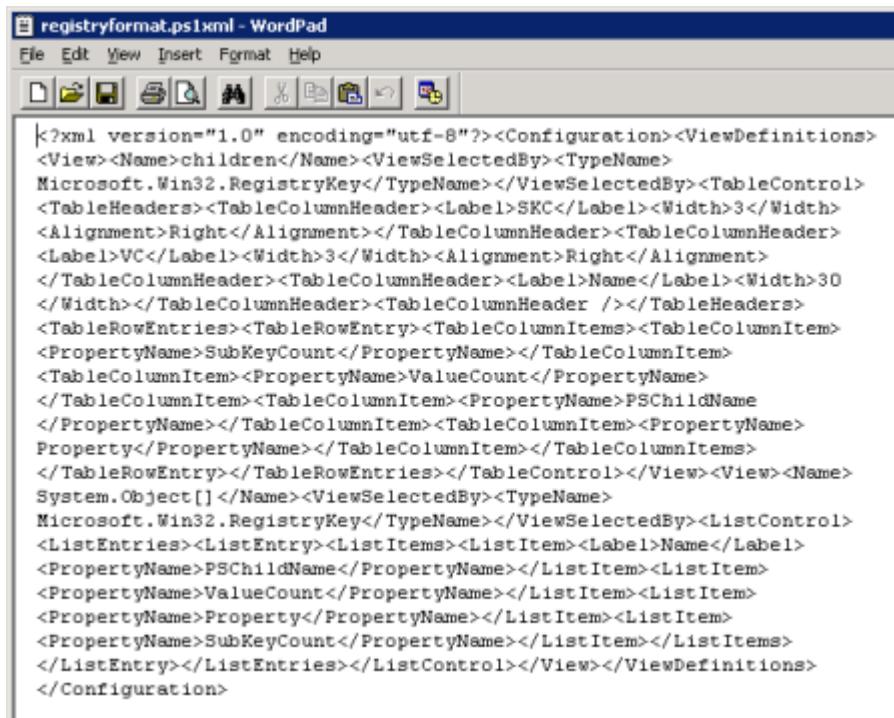
Take formatting data generated by [Get-FormatData](#) and export it to a \*.ps1xml file.

#### Example:

Retrieve the formatting for the TypeName **Microsoft.Win32.RegistryKey** and export it to a \*.ps1xml file.

```
Get-FormatData -TypeName Microsoft.Win32.RegistryKey  
| Export-FormatData -Path registryformat.ps1xml -IncludeScriptBlock
```

The contents of **registryformat.ps1xml** are shown below.



```
<?xml version="1.0" encoding="utf-8"?><Configuration><ViewDefinitions>  
<View><Name>children</Name><ViewSelectedBy><TypeName>  
Microsoft.Win32.RegistryKey</TypeName></ViewSelectedBy><TableControl>  
<TableHeaders><TableColumnHeader><Label>SKC</Label><Width>3</Width>  
<Alignment>Right</Alignment></TableColumnHeader><TableColumnHeader>  
<Label>VC</Label><Width>3</Width><Alignment>Right</Alignment>  
</TableColumnHeader><TableColumnHeader><Label>Name</Label><Width>30  
</Width></TableColumnHeader><TableColumnHeader /></TableHeaders>  
<TableRowEntries><TableRowEntry><TableColumnItems><TableColumnItem>  
<PropertyName>SubKeyCount</PropertyName></TableColumnItem>  
<TableColumnItem><PropertyName>ValueCount</PropertyName>  
</TableColumnItem><TableColumnItem><PropertyName>PSChildName  
</PropertyName></TableColumnItem><TableColumnItem><PropertyName>  
Property</PropertyName></TableColumnItem></TableColumnItems>  
</TableRowEntry></TableRowEntries></TableControl></View><View><Name>  
System.Object[]</Name><ViewSelectedBy><TypeName>  
Microsoft.Win32.RegistryKey</TypeName></ViewSelectedBy><ListControl>  
<ListEntries><ListEntry><ListItems><ListItem><Label>Name</Label>  
<PropertyName>PSChildName</PropertyName></ListItem><ListItem>  
<PropertyName>ValueCount</PropertyName></ListItem><ListItem>  
<PropertyName>Property</PropertyName></ListItem><ListItem>  
<PropertyName>SubKeyCount</PropertyName></ListItem></ListItems>  
</ListEntry></ListEntries></ListControl></View></ViewDefinitions>  
</Configuration>
```

#### How could I have done this in PowerShell 1.0?

You would have needed to manually create your own \*.ps1xml files.

#### Related Cmdlets

##### [Get-FormatData](#)

[Update-FormatData](#)

## #89 Invoke-WmiMethod

### [Invoke-WmiMethod](#)

#### What can I do with it?

Call WMI methods.

#### Example:

Retrieve the WMI instances of the **Print Spooler** service. Pipe it through to **Invoke-WmiMethod** and call the **StopService** method.

```
Get-WmiObject Win32_Service -Filter "name='spooler'"  
| Invoke-WmiMethod -Name StopService
```

Notice the service **State** changes.

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. It displays three commands related to the 'spooler' service:

- The first command, 'Get-WmiObject Win32\_Service -Filter "name='spooler'", shows the service is running with a status of 'OK'.
- The second command, 'Get-WmiObject win32\_service -filter "name='spooler'" | Invoke-WmiMethod -Name StopService', uses the pipeline operator to pass the service object to the 'Invoke-WmiMethod' cmdlet, which calls the 'StopService' method.
- The third command, 'Get-WmiObject Win32\_Service -filter "name='spooler'"', shows the service is now stopped with a status of 'OK'.

In the middle command, the 'State' property is highlighted with a red oval, and its value 'Running' is circled in red. In the bottom command, the 'State' property is also highlighted with a red oval, and its value 'Stopped' is circled in red.

#### How could I have done this in PowerShell 1.0?

In the above example you could have called the **StopService** method on the object returned in the WMI query. One of the advantages though of **Invoke-WmiMethod** is the possibility to use the pipeline.

```
(Get-WmiObject Win32_Service -filter "name='spooler!").StopService()
```

#### Related Cmdlets

[Get-WmiObject](#)

[Remove-WmiObject](#)

[Set-WmiInstance](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-WSManInstance](#)

[Remove-WSManInstance](#)

## #90 Remove-WmiObject

[Remove-WmiObject](#)

### What can I do with it?

Remove an instance of a WMI class.

#### Example:

Start Windows Calculator. Retrieve the running process and terminate it.

```
calc  
Get-WmiObject Win32_Process -Filter "name='calc.exe'" | Remove-WmiObject
```

### How could I have done this in PowerShell 1.0?

In the above example you could have called the **Terminate** method on the object returned in the WMI query. One of the advantages though of **Remove-WmiObject** is the possibility to use the pipeline.

```
(Get-WmiObject Win32_Process -Filter "name='calc.exe'").Terminate()
```

### Related Cmdlets

[Get-WmiObject](#)

[Invoke-WmiMethod](#)

[Set-WmiInstance](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-WSManInstance](#)

[Remove-WSManInstance](#)

## #91 Set-WmiInstance

### Set-WmiInstance

#### What can I do with it?

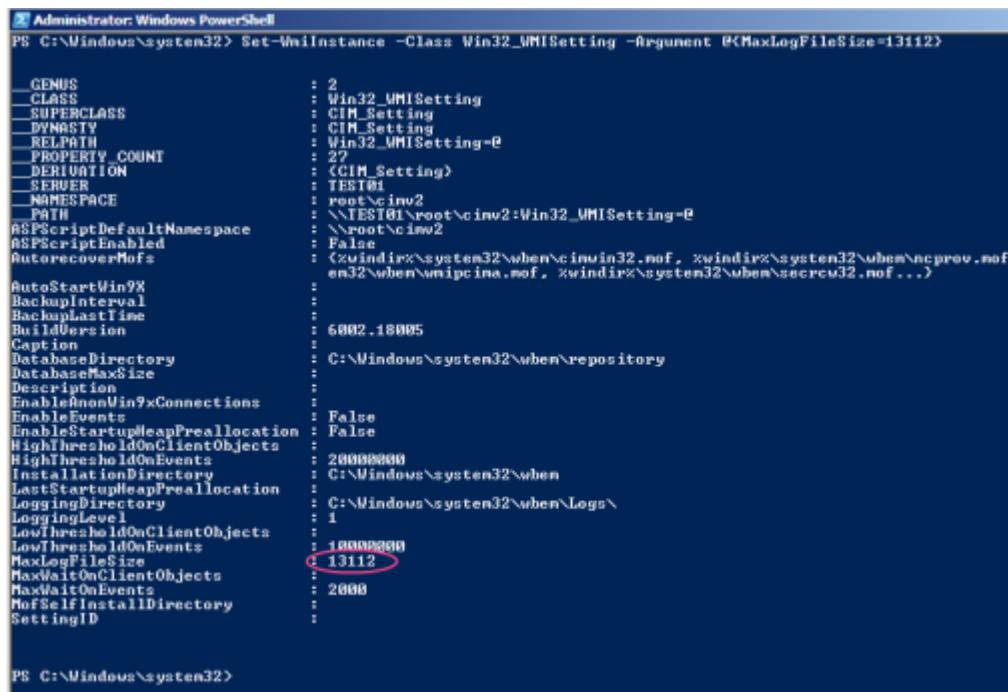
Set an instance of a WMI class.

#### Example:

Change the value of **MaxLogFileSize** within the **Win32\_WMISetting** class from the default of **6556** to **13112**.

```
Set-WmiInstance -Class Win32_WMISetting  
-Argument @{MaxLogFileSize=13112}
```

You will notice that the **MaxLogFileSize** value is updated:



```
Administrator: Windows PowerShell
PS C:\Windows\system32> Set-WmiInstance -Class Win32_WMISetting -Argument @{MaxLogFileSize=13112}

--GENUS
--CLASS
--SUPERCLASS
--DYNASTY
--RELPATH
--PROPERTY_COUNT
--DERIVATION
--SERVER
--NAMESPACE
--PATH
$PScriptDefaultNamespace
$PScriptEnabled
$recoverMof
--AutoStartWin9x
BackupInterval
BackupLastTime
BuildVersion
Caption
DatabaseDirectory
DatabaseMaxSize
Description
EnableNonWin9xConnections
EnableEvents
EnableStartupHeapPreallocation
HighThresholdOnClientObjects
HighThresholdOnEvents
InstallationDirectory
LastStartupHeapPreallocation
LoggingDirectory
LoggingLevel
LowThresholdOnClientObjects
LowThresholdOnEvents
MaxLogFileSize
MaxWaitOnClientObjects
MaxWaitOnEvents
MofSelfInstallDirectory
SettingID
-- : 2
-- : Win32_WMISetting
-- : CIM_Setting
-- : CIM_Setting
-- : Win32_WMISetting@0
-- : 22
-- : <CIM_Setting>
-- : TEST#1
-- : root\cimv2
-- : \TEST#1\root\cimv2:Win32_WMISetting@0
-- : \root\cimv2
-- : False
-- : <\windir\system32\wbem\cimwin32.mof, \windir\system32\wbem\ncprov.mof,
-- : en32\wbem\wmpcima.mof, \windir\system32\wbem\secrev32.mof...>
-- : 6002.18005
-- : C:\Windows\system32\wbem\Repository
-- : 20000000
-- : C:\Windows\system32\wbem
-- : C:\Windows\system32\wbem\Logo\
-- : 1
-- : 10000000
-- : 13112
-- : 2000
-- : 1
-- : 
```

#### How could I have done this in PowerShell 1.0?

In the above example you could have called the **Put** method on the object returned in the WMI query.

```
$wmisetting = Get-WmiObject Win32_WMISetting
$wmisetting.MaxLogFileSize = 13112
$wmisetting.Put()
```

#### Related Cmdlets

### Get-WmiObject

[Invoke-WmiMethod](#)

[Remove-WmiObject](#)

[Get-WSManInstance](#)

[Invoke-WSManAction](#)

[New-WSManInstance](#)

[Remove-WSManInstance](#)

## #92 Register-WmiEvent

### Register-WmiEvent

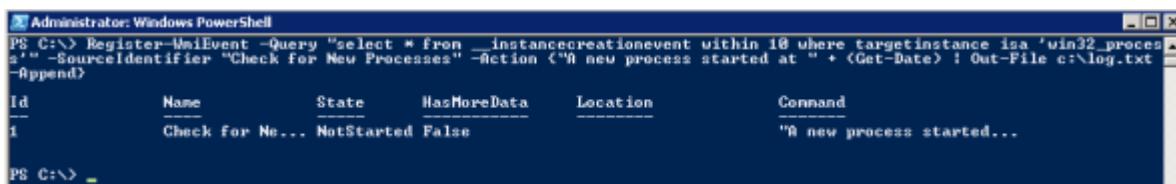
#### What can I do with it?

Subscribe to a WMI event on a local or remote computer and carry out actions based on the event.

#### Example:

Register for a WMI which checks every 10 seconds for any new processes which have started, call it **Check for New Processes** and save information including the date and time out to a log file.

```
Register-WmiEvent -Query "select * from __instancecreationevent within 10
where targetinstance isa 'win32_process'"
-SourcIdentifier "Check for New Processes"
-Action {"A new process started at " + (Get-Date) | Out-File c:\log.txt -
Append}
```



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered is:

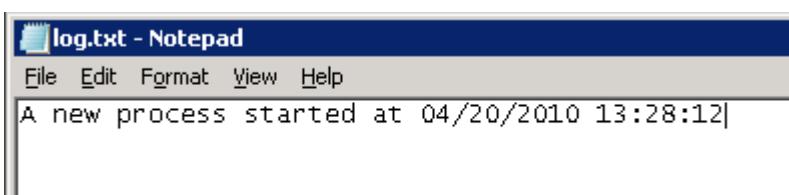
```
PS C:\> Register-WmiEvent -Query "select * from __instancecreationevent within 10 where targetinstance isa 'win32_process'" -SourcIdentifier "Check for New Processes" -Action {"A new process started at " + (Get-Date) : Out-File c:\log.txt -Append}
```

The output shows a table with one row:

Id	Name	State	HasMoreData	Location	Command
1	Check for Ne...	NotStarted	False		"A new process started...

PS C:\>

After running the above command and then starting a process the below is automatically written to **c:\log.txt** after a few seconds.



#### How could I have done this in PowerShell 1.0?

The Scripting Guys detail how to do this in PowerShell 1.0 in this [article](#) by using .NET. The code to achieve it is reproduced below:

```
$a = 0

$timespan = New-Object System.TimeSpan(0, 0, 1)
$scope = New-Object System.Management.ManagementScope("\\.\root\cimV2")
$query = New-Object System.Management.WQLEventQuery `(
    ("__InstanceDeletionEvent", $timespan, "TargetInstance ISA
'Win32_Process' ")
$watcher = New-Object
```

```
System.Management.ManagementEventWatcher($scope,$query)

do
{
    $b = $watcher.WaitForNextEvent()
    $b.TargetInstance.Name
}
while ($a -ne 1)
```

## Related Cmdlets

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #93 Register-ObjectEvent

### Register-ObjectEvent

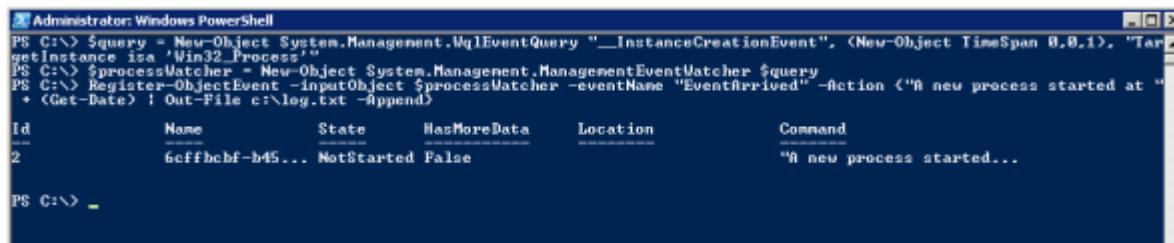
#### What can I do with it?

Subscribe to an event on a local or remote computer generated by a .NET Framework object and carry out actions based on the event.

#### Example:

Register for an event to check for new processes, use the [ManagementEventWatcher](#) .NET object to form the basis of the object to monitor and save information including the date and time out to a log file.

```
$query = New-Object System.Management.WqlEventQuery  
"__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa  
'Win32_Process'"  
$processWatcher = New-Object System.Management.ManagementEventWatcher  
$query  
Register-ObjectEvent -InputObject $processWatcher -EventName "EventArrived"  
-Action {"A new process started at " + (Get-Date) | Out-File c:\log.txt -  
Append}
```



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command history is as follows:

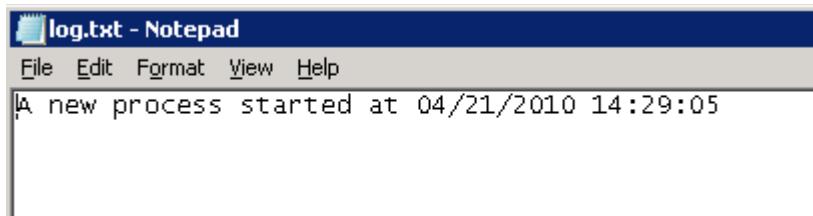
```
PS C:\> $query = New-Object System.Management.WqlEventQuery "__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa  
'Win32_Process'"  
PS C:\> $processWatcher = New-Object System.Management.ManagementEventWatcher $query  
PS C:\> Register-ObjectEvent -InputObject $processWatcher -EventName "EventArrived" -Action {"A new process started at "  
+ (Get-Date) | Out-File c:\log.txt -Append}
```

A table is displayed showing the registered event:

ID	Name	State	HasMoreData	Location	Command
2	6cffbcbf-bd5...	NotStarted	False		"A new process started..."

PS C:\> \_

After running the above commands and then starting a process the below is automatically written to **c:\log.txt** after a few seconds.



#### How could I have done this in PowerShell 1.0?

The Scripting Guys detail how to do this in PowerShell 1.0 in this [article](#) by using .NET. The code to achieve it is reproduced below:

```
$a = 0
```

```

$timespan = New-Object System.TimeSpan(0, 0, 1)
$scope = New-Object System.Management.ManagementScope("\\.\root\cimV2")
$query = New-Object System.Management.WQLEventQuery ` 
    ("__InstanceDeletionEvent",$timespan, "TargetInstance ISA
'Win32_Process' ")
$watcher = New-Object
System.Management.ManagementEventWatcher($scope,$query)

do
{
    $b = $watcher.WaitForNextEvent()
    $b.TargetInstance.Name
}
while ($a -ne 1)

```

## **Related Cmdlets**

[Register-WmiEvent](#)

[Register-EngineEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #94 Get-EventSubscriber

### [Get-EventSubscriber](#)

#### What can I do with it?

Retrieve event subscribers from the current session.

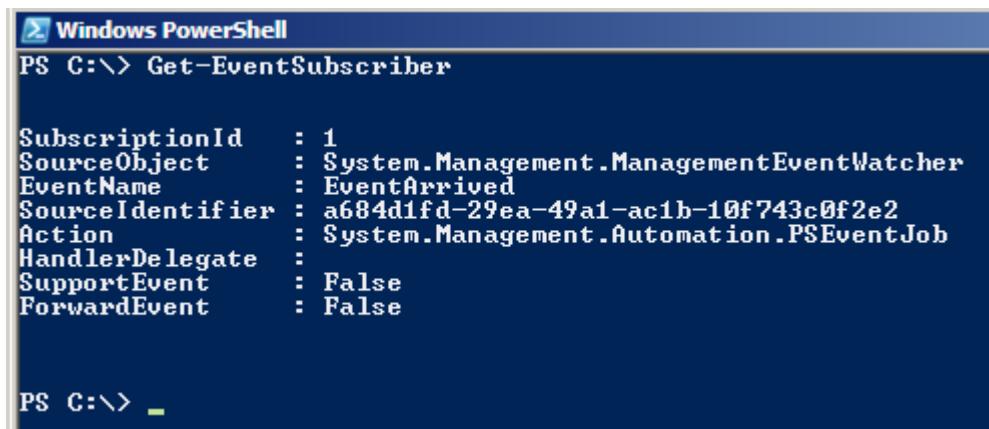
#### Example:

Use the [Register-ObjectEvent](#) cmdlet to register for an event to check for new processes, use the [ManagementEventWatcher](#) .NET object to form the basis of the object to monitor and save information including the date and time out to a log file.

Execution of Get-EventSubscriber will then show details of the event subscribed to.

```
$query = New-Object System.Management.WqlEventQuery
"__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa
'Win32_Process'"
$processWatcher = New-Object System.Management.ManagementEventWatcher
$query
Register-ObjectEvent -InputObject $processWatcher -EventName "EventArrived"
-Action {"A new process started at " + (Get-Date) | Out-File c:\log.txt -
Append}
Get-EventSubscriber
```

You will see below the details which are returned by default



```
Windows PowerShell
PS C:\> Get-EventSubscriber

SubscriptionId    : 1
SourceObject      : System.Management.ManagementEventWatcher
EventName         : EventArrived
SourceIdentifier  : a684d1fd-29ea-49a1-ac1b-10f743c0f2e2
Action            : System.Management.Automation.PSEventJob
HandlerDelegate   :
SupportEvent      : False
ForwardEvent      : False

PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

[Register-ObjectEvent](#) and [Register-WmiEvent](#) contain examples of how to create events in .NET

#### Related Cmdlets

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #95 Register-EngineEvent

### [Register-EngineEvent](#)

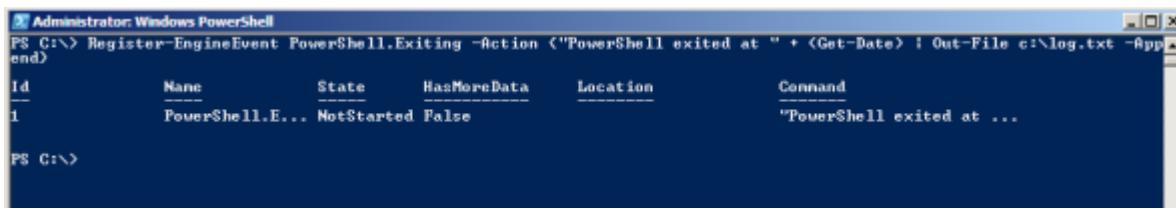
#### What can I do with it?

Subscribe to events generated by the PowerShell engine or the New-Event cmdlet.

#### Example:

Subscribe to an event when the PowerShell session exits, and save information including the date and time out to a log file.

```
Register-EngineEvent PowerShell.Exiting
-Action {"PowerShell exited at " + (Get-Date) | Out-File c:\log.txt -Append}
```

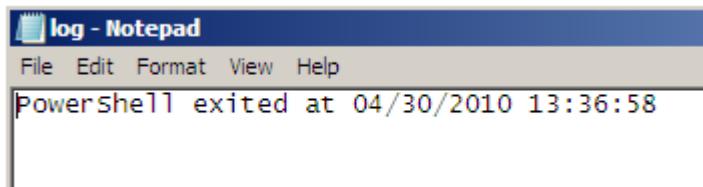


A screenshot of a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is 'Register-EngineEvent PowerShell.Exiting -Action {"PowerShell exited at " + (Get-Date) | Out-File c:\log.txt -Append}'. The output shows a table with one row of data:

Id	Name	State	HasMoreData	Location	Command
1	PowerShell.E...	NotStarted	False		"PowerShell exited at ..."

PS C:\>

After closing the PowerShell session (by typing **Exit**) the date and time it was closed is written to the log file.



#### How could I have done this in PowerShell 1.0?

PowerShell engine events are a new feature in PowerShell 2.0.

#### Related Cmdlets

[Register-ObjectEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #96 New-Event

### New-Event

#### What can I do with it?

Create a custom event.

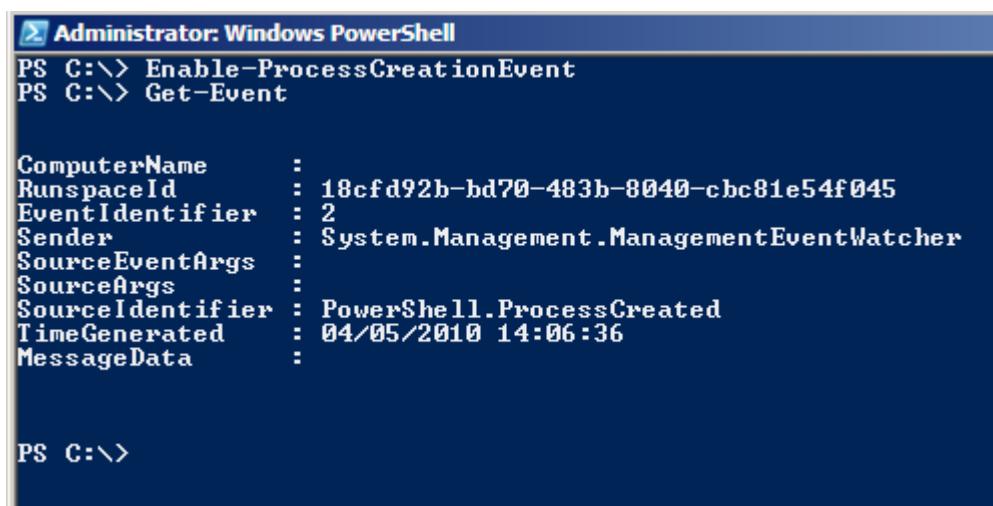
#### Example:

The built-in PowerShell help has a great example for New-Event. It uses New-Event to create a custom event based on a reaction to another event.

```
function Enable-ProcessCreationEvent
{
    $query = New-Object System.Management.WqlEventQuery
    "$_InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa
    'Win32_Process'"
    $processWatcher = New-Object System.Management.ManagementEventWatcher
    $query
    $identifier = "WMI.ProcessCreated"

    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
    $identifier -Action {
        [void] (New-Event -SourceIdentifier "PowerShell.ProcessCreated" -
        Sender $args[0] -EventArgs
        $args[1].SourceEventArgs.NewEvent.TargetInstance)
    }
}
```

You will notice that if you execute this function and then create a new process a new event is automatically generated.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Enable-ProcessCreationEvent is run, followed by PS C:\> Get-Event. The output of Get-Event is displayed as a table:

	:
ComputerName	:
RunspaceId	: 18cf92b-bd70-483b-8040-cbc81e54f045
EventIdentifier	: 2
Sender	: System.Management.ManagementEventWatcher
SourceEventArgs	:
SourceArgs	:
SourceIdentifier	: PowerShell.ProcessCreated
TimeGenerated	: 04/05/2010 14:06:36
MessageData	:

PS C:\>

#### How could I have done this in PowerShell 1.0?

PowerShell engine events are a new feature in PowerShell 2.0.

## **Related Cmdlets**

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #97 Get-Event

### Get-Event

#### What can I do with it?

Retrieve events from the event queue.

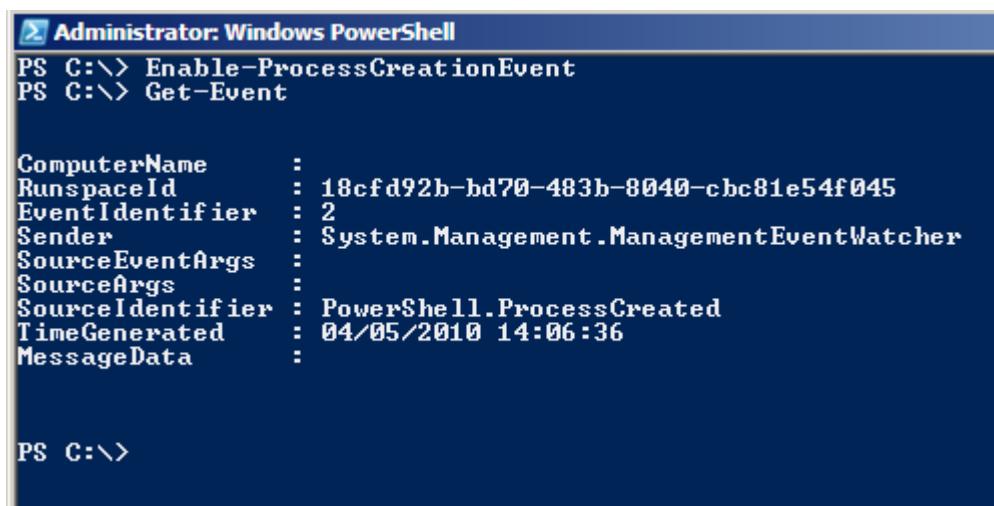
#### Example:

The built-in PowerShell help has a great example for [New-Event](#). It uses [New-Event](#) to create a custom event based on a reaction to another event. Once the event has been created Get-Event can be used to examine details of that event and any others currently in the queue.

```
function Enable-ProcessCreationEvent
{
    $query = New-Object System.Management.WqlEventQuery
"__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa
'Win32_Process'"
    $processWatcher = New-Object System.Management.ManagementEventWatcher
$query
    $identifier = "WMI.ProcessCreated"

    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
$identifier -Action {
    [void] (New-Event -SourceIdentifier "PowerShell.ProcessCreated" -
Sender $args[0] -EventArgs
$args[1].SourceEventArgs.NewEvent.TargetInstance)
}
}
Get-Event
```

You will notice that if you execute this function and then create a new process a new event is automatically generated. Get-Event retrieves details of this event.



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Enable-ProcessCreationEvent" is run first, followed by "Get-Event". The output shows a single event object with the following properties:

ComputerName	:	
RunspaceId	:	18cf92b-bd70-483b-8040-cbc81e54f045
EventIdentifier	:	2
Sender	:	System.Management.ManagementEventWatcher
SourceEventArgs	:	
SourceArgs	:	
SourceIdentifier	:	PowerShell.ProcessCreated
TimeGenerated	:	04/05/2010 14:06:36
MessageData	:	

PS C:\>

## **How could I have done this in PowerShell 1.0?**

PowerShell engine events are a new feature in PowerShell 2.0.

### **Related Cmdlets**

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #98 Wait-Event

### Wait-Event

#### What can I do with it?

Pause a running script or session and wait for an event to occur before continuing.

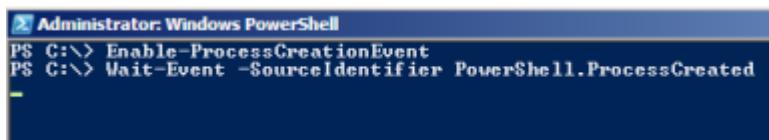
#### Example:

The built-in PowerShell help has a great example for [New-Event](#). It uses [New-Event](#) to create a custom event based on a reaction to another event. Use Wait-Event to make the current session pause until a new process has been opened. Open Windows Calculator to make the event trigger and return the prompt to the user.

```
function Enable-ProcessCreationEvent
{
    $query = New-Object System.Management.WqlEventQuery
"__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa
'Win32_Process'"
    $processWatcher = New-Object System.Management.ManagementEventWatcher
$query
    $identifier = "WMI.ProcessCreated"

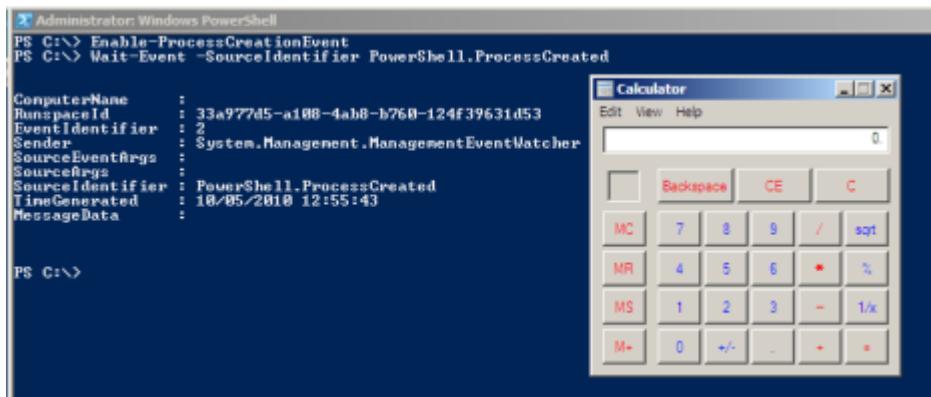
    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
$identifier -Action {
        [void] (New-Event -SourceIdentifier "PowerShell.ProcessCreated" -
Sender $args[0] -EventArgs
$args[1].SourceEventArgs.NewEventArgs.TargetInstance)
    }
}
Wait-Event -SourceIdentifier PowerShell.ProcessCreated
```

Before opening Windows Calculator:



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Enable-ProcessCreationEvent is entered, followed by PS C:\> Wait-Event -SourceIdentifier PowerShell.ProcessCreated. The window shows a single character '-' at the bottom, indicating the command is still executing.

You will notice that after the opening of Windows Calculator the event is triggered and the prompt is returned to the user.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell" running on Windows 7. The command PS C:\> Enable-ProcessCreationEvent is run, followed by PS C:\> Wait-Event -SourceIdentifier PowerShell.ProcessCreated. This command monitors for new process creation events. A second PowerShell window shows the results of the event capture, displaying properties like ComputerName, RunspaceId, EventIdentifier, Sender, SourceEventArgs, SourceIdentifier, TimeGenerated, and MessageData. Below the command line, PS C:\> is visible. To the right of the PowerShell windows, a standard Windows calculator application is open.

```
PS C:\> Enable-ProcessCreationEvent
PS C:\> Wait-Event -SourceIdentifier PowerShell.ProcessCreated

ComputerName : 
RunspaceId   : 33a97745-a108-4ab8-b760-124f39631d53
EventIdentifier : 2
Sender       : System.Management.ManagementEventWatcher
SourceEventArgs :
SourceIdentifier : PowerShell.ProcessCreated
TimeGenerated  : 10/05/2010 12:55:43
MessageData   :

PS C:\>
```

## How could I have done this in PowerShell 1.0?

PowerShell engine events are a new feature in PowerShell 2.0.

### Related Cmdlets

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

## #99 Unregister-Event

[Unregister-Event](#)

### What can I do with it?

Clear an event subscription.

#### Example:

Use [Get-EventSubscriber](#) to retrieve details of current events. Clear the event with subscription id 1 and [Get-EventSubscriber](#) again to confirm that it has been removed.

```
Get-EventSubscriber  
Unregister-Event -SubscriptionId 1  
Get-EventSubscriber
```

You will see that the event subscription has been cleared.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-EventSubscriber" is run, displaying a single subscription with SubscriptionId 1. Then, "Unregister-Event -SubscriptionId 1" is run, followed by another "Get-EventSubscriber" which shows no results, indicating the subscription has been removed.

```
PS C:\> Get-EventSubscriber  
SubscriptionId : 1  
SourceObject   : System.Management.ManagementEventWatcher  
EventName      : EventArrived  
SourceIdentifier : 78528dbb-ab2a-466c-bd81-1c07b3636f52  
Action         : System.Management.Automation.PSEventJob  
HandlerDelegate :  
SupportEvent    : False  
ForwardEvent    : False  
  
PS C:\> Unregister-Event -SubscriptionId 1  
PS C:\> Get-EventSubscriber  
PS C:\> -
```

### How could I have done this in PowerShell 1.0?

PowerShell engine events are a new feature in PowerShell 2.0.

### Related Cmdlets

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Get-Event](#)

[New-Event](#)

[Remove-Event](#)

[Wait-Event](#)

## #100 Remove-Event

### Remove-Event

#### What can I do with it?

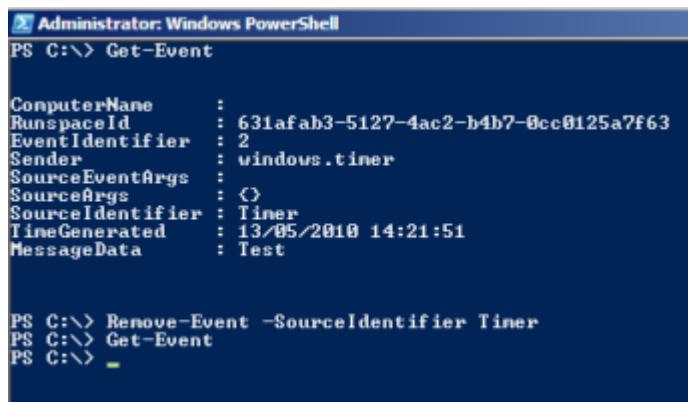
Delete an event from the current session. **Note:** to unsubscribe from an event you will need to use [Unregister-Event](#).

#### Example:

Retrieve current events in the queue with [Get-Event](#), use Remove-Event to clear the event with the SourceIdentifier of **Timer**, then [Get-Event](#) again to confirm that it has been removed.

```
Get-Event  
Remove-Event -SourceIdentifier Timer  
Get-Event
```

You will see that the event has been cleared.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "Get-Event" is run, displaying a single event object with properties like ComputerName, RunspaceId, EventIdentifier, Sender, SourceEventArgs, SourceArgs, SourceIdentifier, TimeGenerated, and MessageData. The event identifier is 2 and the source identifier is "Timer". The command "Remove-Event -SourceIdentifier Timer" is then run, followed by another "Get-Event" command which returns no results, indicating the event has been removed.

```
Administrator: Windows PowerShell  
PS C:\> Get-Event  
  
ComputerName      :  
RunspaceId        : 631afab3-5127-4ac2-b4b7-0cc0125a7f63  
EventIdentifier   : 2  
Sender            : windows.timer  
SourceEventArgs  :  
SourceArgs        : ()  
SourceIdentifier  : Timer  
TimeGenerated     : 13/05/2010 14:21:51  
MessageData       : Test  
  
PS C:\> Remove-Event -SourceIdentifier Timer  
PS C:\> Get-Event  
PS C:\> _
```

#### How could I have done this in PowerShell 1.0?

PowerShell engine events are a new feature in PowerShell 2.0.

#### Related Cmdlets

[Register-ObjectEvent](#)

[Register-EngineEvent](#)

[Register-WmiEvent](#)

[Unregister-Event](#)

[Get-Event](#)

[New-Event](#)

## Wait-Event

## #101 Wait-Process

### Wait-Process

#### What can I do with it?

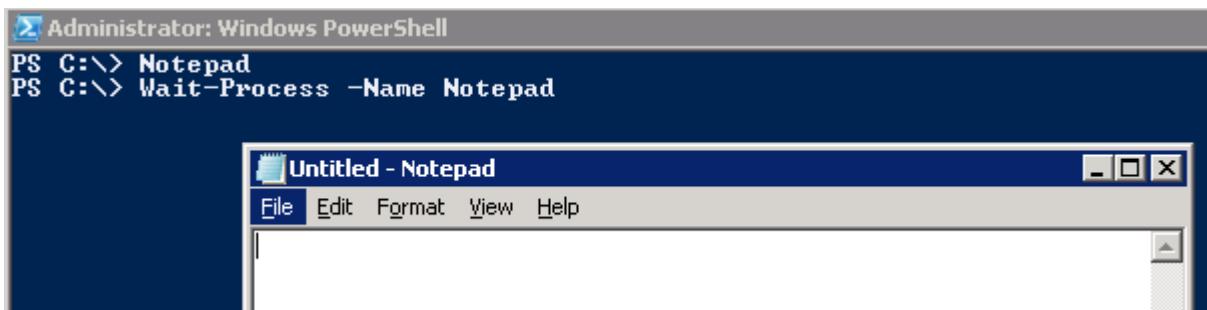
Wait for a process to stop before proceeding further.

#### Example:

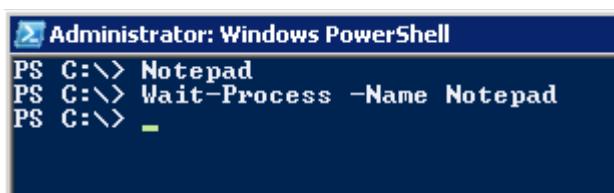
Open an instance of Notepad. Use Wait-Process to pause the console session until Notepad is closed.

```
Notepad  
Wait-Process -Name Notepad
```

You will notice that the console pauses whilst Notepad is open



Once Notepad is closed, control of the session is returned to the user.



#### How could I have done this in PowerShell 1.0?

Store the result of Get-Process Notepad in a variable, then use the **WaitForExit** method to wait for the process to stop.

```
Notepad  
$Process = Get-Process Notepad  
$Process.WaitForExit()
```

#### Related Cmdlets

[Get-Process](#)

[Start-Process](#)

Stop-Process

Debug-Process

## #102 Disable-PSRemoting

[Disable-PSRemoting](#). Note: This is a proxy command which calls the [Disable-PSSessionConfiguration](#) cmdlet.

### What can I do with it?

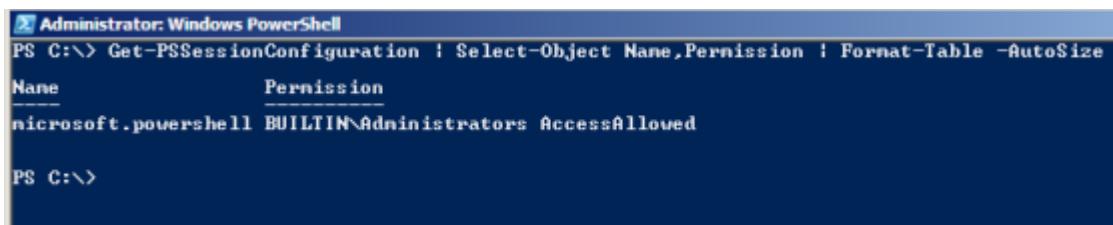
Disable PowerShell remoting on a computer that has previously been enabled for remoting.  
**Note:** This command must be run from a PowerShell session with administrative privileges.

#### Example:

Retrieve the current PSSessionConfiguration settings. Disable PowerShell remoting, and then retrieve the PSSessionConfiguration settings again to compare.

```
Get-PSSessionConfiguration  
Disable-PSRemoting  
Get-PSSessionConfiguration
```

Notice the PSSessionConfiguration on a machine enabled for PowerShell remoting



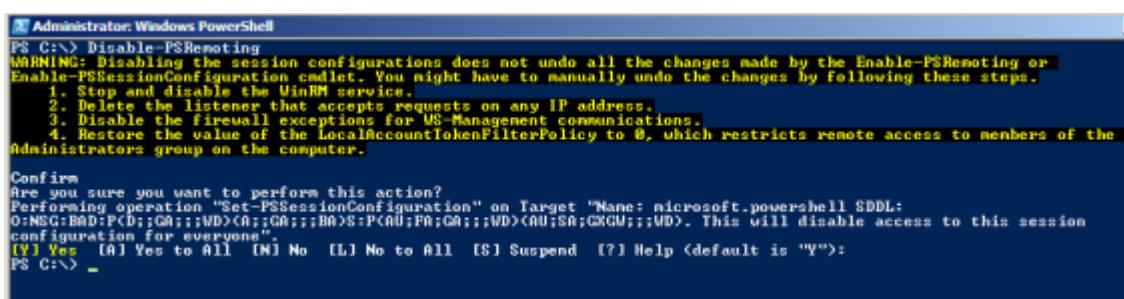
```
Administrator: Windows PowerShell  
PS C:\> Get-PSSessionConfiguration | Select-Object Name,Permission | Format-Table -AutoSize  
Name          Permission  
microsoft.powershell BUILTIN\Administrators AccessAllowed  
PS C:\>
```

Disable PowerShell Remoting. You will receive the following warning that using this command does not necessarily reverse everything that Enable-PSRemoting may have done.

**WARNING:** Disabling the session configurations does not undo all the changes made by the Enable-PSRemoting or

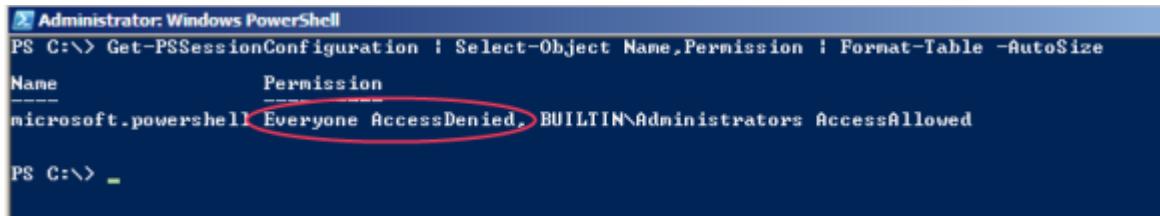
Enable-PSSessionConfiguration cmdlet. You might have to manually undo the changes by following these steps.

1. Stop and disable the WinRM service.
2. Delete the listener that accepts requests on any IP address.
3. Disable the firewall exceptions for WS-Management communications.
4. Restore the value of the LocalAccountTokenFilterPolicy to 0, which restricts remote access to members of the Administrators group on the computer.



```
Administrator: Windows PowerShell  
PS C:\> Disable-PSRemoting  
WARNING: Disabling the session configurations does not undo all the changes made by the Enable-PSRemoting or  
Enable-PSSessionConfiguration cmdlet. You might have to manually undo the changes by following these steps.  
1. Stop and disable the WinRM service.  
2. Delete the listener that accepts requests on any IP address.  
3. Disable the firewall exceptions for WS-Management communications.  
4. Restore the value of the LocalAccountTokenFilterPolicy to 0, which restricts remote access to members of the  
Administrators group on the computer.  
Confirm  
Are you sure you want to perform this action?  
Performing operation "Set-PSSessionConfiguration" on Target "Name: microsoft.powershell SDDL:  
O:NG:BADP(CD;GA;;WD)<A;;GA;;BA>S:P(AU;PA;GA;;WD)<AU;SA;GXGW;;WD). This will disable access to this session  
configuration for everyone".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help <default is "Y">  
PS C:\>
```

Now check the impact on the PSSessionconfiguration - the **AccessDenied** permission has been applied to **Everyone**.



Name	Permission
microsoft.powershell	Everyone AccessDenied, BUILTIN\Administrators AccessAllowed

### How could I have done this in PowerShell 1.0?

Remoting did not exist in PowerShell 1.0; you would have needed to use Remote Desktop to run an interactive session on a remote server.

### Related Cmdlets

[Enable-PSRemoting](#)

[Disable-PSSessionConfiguration](#)

[Get-PSSessionConfiguration](#)

[Register-PSSessionConfiguration](#)

[Set-PSSessionConfiguration](#)

[Unregister-PSSessionConfiguration](#)

## #103 Update-List

### Update-List

#### What can I do with it?

Add, Remove or Replace items from a property value of an object. This cmdlet can only update a property when it supports the  [IList interface](#). So far this does not include any of the core Windows PowerShell cmdlets - however it does include some of the cmdlets that ship with Exchange 2007 and later.

#### Example:

Add additional email addresses to the Test1 user's mailbox using the **Add** parameter of Update-List.

```
Get-Mailbox Test1 | Update-List -Property EmailAddresses -Add  
admin@contoso.com, webmaster@contoso.com  
| Set-Mailbox
```

#### How could I have done this in PowerShell 1.0?

Shay Levy has a great [blog post](#) on dealing with AD / Mailbox accounts with multi-valued attributes.

## #104 Trace-Command

### Trace-Command

#### What can I do with it?

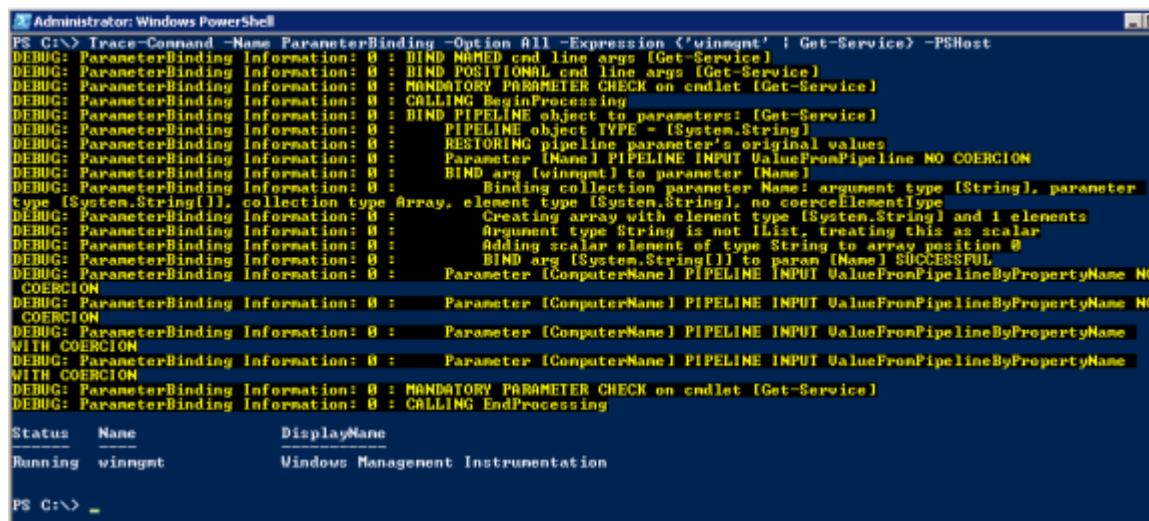
Begin a trace of a command or expression.

#### Example:

Examine debug info for Parameter Binding when piping a string through to Get-Service.

```
Trace-Command -Name ParameterBinding -Option All -Expression {'winmgmt' | Get-Service} -PSHost
```

You will see it is possible to work through the debug info to find out what is happening:



```
Administrator: Windows PowerShell
PS C:\> Trace-Command -Name ParameterBinding -Option All -Expression {'winmgmt' | Get-Service} -PSHost
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Get-Service]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Get-Service]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Service]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Get-Service]
DEBUG: ParameterBinding Information: 0 : PIPELINE object TYPE = [System.String]
DEBUG: ParameterBinding Information: 0 : RESTORING pipeline parameter's original values
DEBUG: ParameterBinding Information: 0 : Parameter [Name] PIPELINE INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 : BIND arg [winmgmt] to parameter [Name]
DEBUG: ParameterBinding Information: 0 : Binding collection parameter Name: argument type [String], parameter type [System.String[]], collection type Array, element type [System.String], no coerceElementType
DEBUG: ParameterBinding Information: 0 : Creating array with element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 : Argument type String is not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 : Adding scalar element of type String to array position 0
DEBUG: ParameterBinding Information: 0 : BIND arg [System.String[]] to param [Name] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : Parameter [ComputerName] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION
DEBUG: ParameterBinding Information: 0 : Parameter [ComputerName] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION
DEBUG: ParameterBinding Information: 0 : Parameter [ComputerName] PIPELINE INPUT ValueFromPipelineByPropertyName WITH COERCION
DEBUG: ParameterBinding Information: 0 : Parameter [ComputerName] PIPELINE INPUT ValueFromPipelineByPropertyName WITH COERCION
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Service]
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing

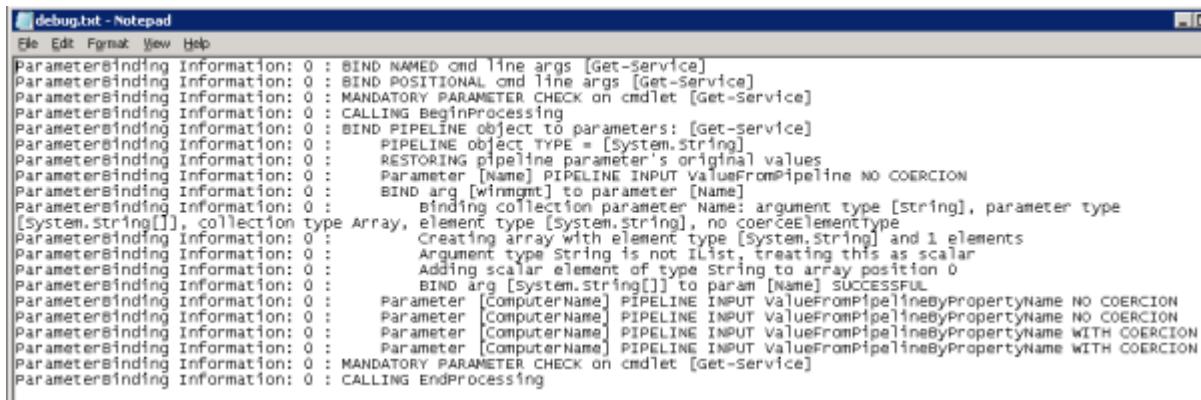
Status      Name          DisplayName
Running    winmgmt       Windows Management Instrumentation

PS C:\> _
```

Note: it is also possible to output the debug info to a file, simply remove the **PSHost** parameter and use **FilePath** instead.

```
Trace-Command -Name ParameterBinding -Option All -Expression {'winmgmt' | Get-Service} -FilePath C:\Debug.txt
```

The resulting debug info is now easily viewable in Notepad.



```
debug.txt - Notepad
File Edit Format View Help
ParameterBinding Information: 0 : BIND NAMED cmd line args [Get-Service]
ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Get-Service]
ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Service]
ParameterBinding Information: 0 : CALLING BeginProcessing
ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Get-Service]
ParameterBinding Information: 0 :      PIPELINE object TYPE = [System.String]
ParameterBinding Information: 0 :      RESTORING pipeline parameter's original values
ParameterBinding Information: 0 :      Parameter [Name] PIPELINE INPUT valueFromPipeline NO COERCION
ParameterBinding Information: 0 :      BIND arg [winmgmt] to parameter [Name]
ParameterBinding Information: 0 :          Binding collection parameter Name: argument type [string], parameter type
[System.String[]], collection type Array, element type [System.String], no coercelementtype
ParameterBinding Information: 0 :          Creating array with element type [System.String] and 1 elements
ParameterBinding Information: 0 :          Argument type String is not IList, treating this as scalar
ParameterBinding Information: 0 :          Adding scalar element of type String to array position 0
ParameterBinding Information: 0 :      BIND arg [System.String[]] to param [Name] SUCCESSFUL
ParameterBinding Information: 0 :      Parameter [ComputerName] PIPELINE INPUT valueFromPipelineByPropertyName NO COERCION
ParameterBinding Information: 0 :      Parameter [ComputerName] PIPELINE INPUT valueFromPipelineByPropertyName NO COERCION
ParameterBinding Information: 0 :      Parameter [ComputerName] PIPELINE INPUT valueFromPipelineByPropertyName WITH COERCION
ParameterBinding Information: 0 :      Parameter [ComputerName] PIPELINE INPUT valueFromPipelineByPropertyName WITH COERCION
ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Service]
ParameterBinding Information: 0 : CALLING EndProcessing
```

## How could I have done this in PowerShell 1.0?

You could have used [Set-TraceSource](#), but Trace-Command applies the trace only to the specified command.

## Related Cmdlets

[Get-TraceSource](#)

[Set-TraceSource](#)

## #105 Set-StrictMode

### Set-StrictMode

#### What can I do with it?

Configure strict mode for the current scope. An error will be generated when the content of an expression, script or script block violates coding rules. Note: it is possible to use the **Version** parameter to pick which coding rules to use. The [PowerShell help](#) lists the current possible options as:

#### 1.0

-- Prohibits references to uninitialized variables, except for uninitialized variables in strings.

#### 2.0

-- Prohibits references to uninitialized variables (including uninitialized variables in strings).

-- Prohibits references to non-existent properties of an object.

-- Prohibits function calls that use the syntax for calling methods.

-- Prohibits a variable without a name (\${}).

#### Latest

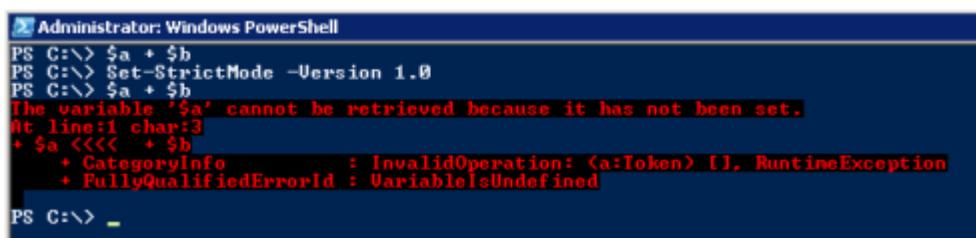
--Selects the latest (most strict) version available. Use this value to assure that scripts use the strictest available version, even when new versions are added to Windows PowerShell.

#### Example:

Examine what happens when you add the undefined \$b to the undefined \$a with strict mode off. Next, turn on strict mode using Version 1.0 and run the same test.

```
$a + $b
Set-StrictMode -Version 1.0
$a + $b
```

Note the error message generated with strict mode on because \$a has not been initialised.



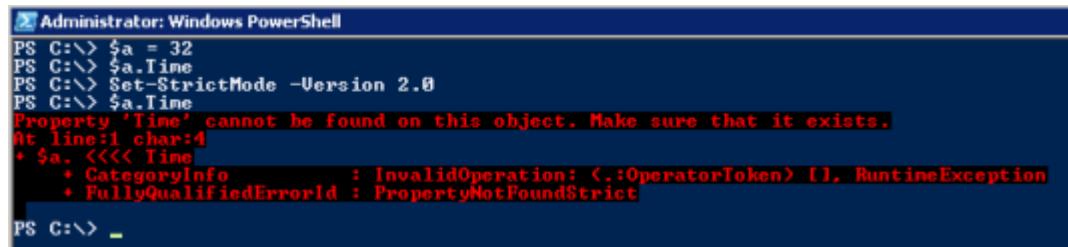
The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was "\$a + \$b" followed by "Set-StrictMode -Version 1.0". The output shows an error message: "The variable '\$a' cannot be retrieved because it has not been set." with additional details about the error type and stack trace.

```
PS C:\> $a + $b
PS C:\> Set-StrictMode -Version 1.0
PS C:\> $a + $b
The variable '$a' cannot be retrieved because it has not been set.
At line:1 char:3
+ $a + $b
+ CategoryInfo          : InvalidOperation: (a:Token) [], RuntimeException
+ FullyQualifiedErrorId : VariableIsUndefined
```

Examine what happens when you define \$a to be a numerical value and attempt to reference a property with strict mode off. Next, turn on strict mode using Version 2.0 and run the same test.

```
$a = 32
$a.Time
Set-StrictMode -Version 2.0
$a.Time
```

Note the error message generated with strict mode on because the Time property does not exist.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command entered was:

```
PS C:\> $a = 32
PS C:\> $a.Time
PS C:\> Set-StrictMode -Version 2.0
PS C:\> $a.Time
Property 'Time' cannot be found on this object. Make sure that it exists.
At line:1 char:4
+ $a. <<< Time
+ CategoryInfo          : InvalidOperation: <.:OperatorToken> [], RuntimeException
+ FullyQualifiedErrorId : PropertyNotFoundStrict
```

The error message is highlighted in red, indicating that the 'Time' property does not exist on the object \$a.

## How could I have done this in PowerShell 1.0?

You could have used [Set-PSDebug](#), however Set-Strictmode applies only to the current scope or child scopes and does not impact the global scope. For more information on scopes in PowerShell [look here](#).

## #106 Import-LocalizedData

### Import-LocalizedData

#### What can I do with it?

Enable text in scripts displayed to users to be presented in their own language. The cmdlet uses the automatic variable **\$PSUICulture** to determine the language to use and alternate text is stored within .psd1 files in subdirectories of the folder that the script is stored.

#### Example:

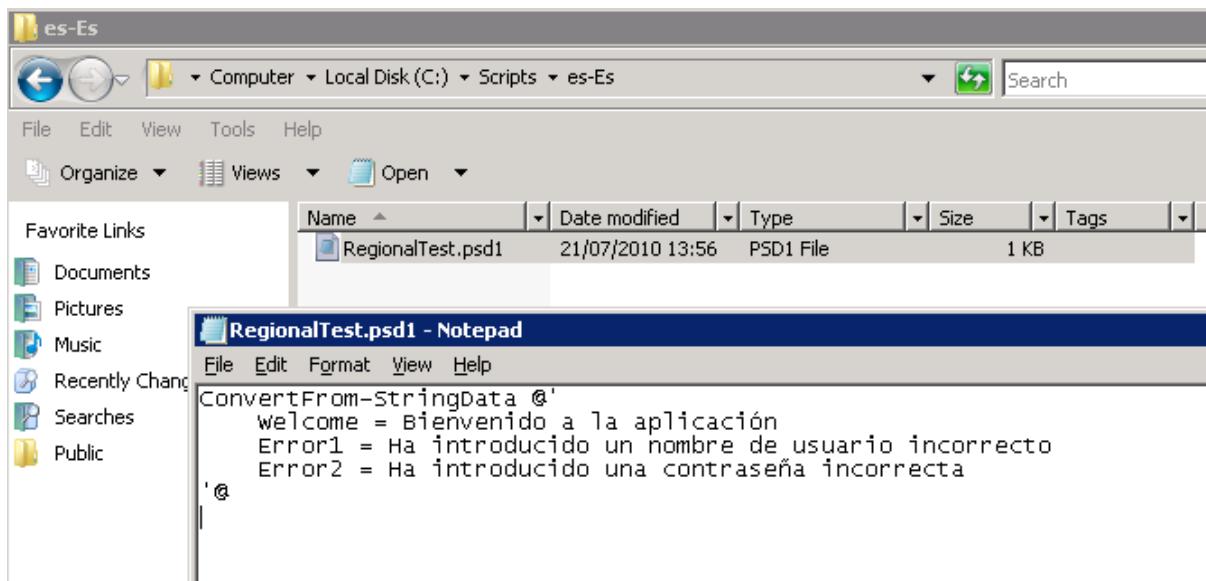
In a script called **RegionalTest.ps1** use the [ConvertFrom-StringData](#) cmdlet to create a series of text messages to display to the user. Import-LocalizedData will retrieve the value of the **\$PSUICulture** automatic variable, get the contents of the **RegionalTest.psd1** file in the es-ES directory (assume the user is Spanish) and store the data within the variable designated by the **BindingVariable** parameter. Then display the Welcome text.

```
$UserMessages = Data {  
    # culture="en-US"  
    ConvertFrom-StringData @'  
        Welcome = Welcome to the application  
        Error1 = You have entered an incorrect username  
        Error2 = You have entered an incorrect password  
'@  
}  
  
Import-LocalizedData -BindingVariable $UserMessages  
  
$UserMessages>Welcome
```

The contents of the RegionalTest.psd1 file for Spanish would look like (apologies for any bad translation!)

```
ConvertFrom-StringData @'  
    Welcome = Bienvenido a la aplicación  
    Error1 = Ha introducido un nombre de usuario incorrecto  
    Error2 = Ha introducido una contraseña incorrecta  
'@
```

and be stored in the es-ES folder below C:\Scripts where RegionalTest.ps1 lives



When run on the Spanish user's machine the Spanish text would be displayed rather than the original English.

### How could I have done this in PowerShell 1.0?

Script Internationalisation features were introduced in PowerShell 2.0 and not supported in version 1.0 - [more info here](#).

## #107 Add-Type

### Add-Type

#### What can I do with it?

Imbed code from modern programming languages into your PowerShell session or scripts. The valid languages are: C#, C# 3.0, VisualBasic and JScript - C# is the default. Use the **Language** parameter to specify one if it is not C#.

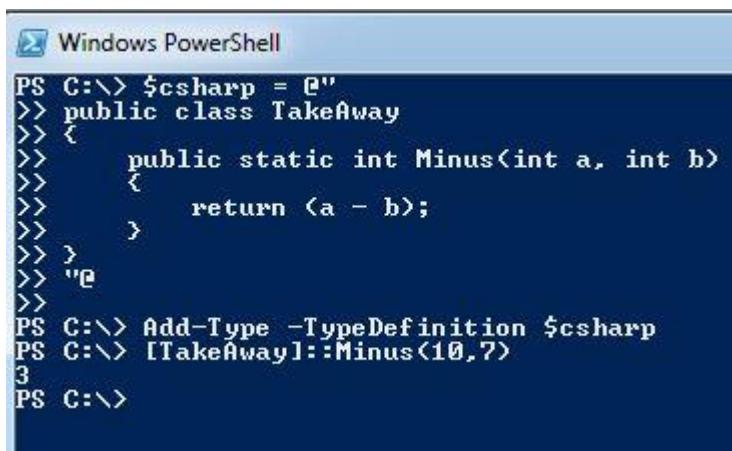
```
-Language <language>
    Specifies the language used in the source code. Add-Type uses the language to select the correct code compiler
    -
    Valid values are "CSharp", "CSharpVersion3", "VisualBasic", and "JScript". "CSharp" is the default.
```

#### Example:

Within a PowerShell session use some C# code to create a **TakeAway** class and create a static method **Minus**. Use the Add-Type cmdlet to add the class to the session and then call the **TakeAway** class and **Minus** static method.

```
$csharp = @"
public class TakeAway
{
    public static int Minus(int a, int b)
    {
        return (a - b);
    }
}
"@
Add-Type -TypeDefinition $csharp
[TakeAway]::Minus(10,7)
```

You will see that we get the expected answer of 3:



```
Windows PowerShell
PS C:\> $csharp = @"
>> public class TakeAway
>> {
>>     public static int Minus(int a, int b)
>>     {
>>         return (a - b);
>>     }
>> }
>> "@
>>
PS C:\> Add-Type -TypeDefinition $csharp
PS C:\> [TakeAway]::Minus(10,7)
3
PS C:\>
```

#### How could I have done this in PowerShell 1.0?

PowerShell 1.0 did not support adding C# or other code into PowerShell scripts, you could however have created your own cmdlet which I'm sure would have been very straightforward

for most sysadmins :-)

## **Related Cmdlets**

[Add-Member](#)

[New-Object](#)