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
<p>| 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 |</p>
<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>#101</td>
<td>Wait-Process</td>
<td>176</td>
</tr>
<tr>
<td>#102</td>
<td>Disable-PSRemoting</td>
<td>178</td>
</tr>
<tr>
<td>#103</td>
<td>Update-List</td>
<td>180</td>
</tr>
<tr>
<td>#104</td>
<td>Trace-Command</td>
<td>181</td>
</tr>
<tr>
<td>#105</td>
<td>Set-StrictMode</td>
<td>183</td>
</tr>
<tr>
<td>#106</td>
<td>Import-LocalizedData</td>
<td>185</td>
</tr>
<tr>
<td>#107</td>
<td>Add-Type</td>
<td>187</td>
</tr>
</tbody>
</table>
#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:**

```powershell
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

```powershell
Function Send-MailMessage () {
    param ($Sender,$Recipient,$Attachment)

    $smtpServer = "smtp.test.local"

    $msg.From = "$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 /\oVV 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:

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'
```
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.

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

The CSV file can be opened in Excel using PowerShell:

```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!

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]$computer = "WinNT://WINDOWS2000/computename$"
$computer.SetPassword("computename$")
```

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

```
ettldom 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

```powershell
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

```powershell
$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
 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 Systernals 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.

```powershell
function Set-Domain {
    param(
        [switch]$help,
        [string]$domain=$($read-host "Please specify the domain to join"),
    )

    $usage = "`$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.

![Image of PowerShell output]

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

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

![Image of PowerShell output]

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
```
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
```

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

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

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

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

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

![Image](https://via.placeholder.com/150)

**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.

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

![Get-Job command to retrieve all jobs and pipe it through to Remove-Job to remove them all.](image)

Use the `Get-Job` cmdlet to retrieve all jobs and pipe it through to Remove-Job to remove them all.

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

![Get-Job command to retrieve all jobs and pipe it through to Remove-Job to remove them all.](image)

**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.

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

Retrieve all current jobs and stop them all.

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

**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.

```
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.
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
<?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"`

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.
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
```

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

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.

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

Restart-Computer
#25 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] = [wmi class]"\\.\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](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.**

```powershell
$weather = New-WebServiceProxy -URI "http://www.webservicex.net/globalweather.asmx?wsdl"
$weather.GetWeather('Southampton', 'United Kingdom')
```

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

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

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

```powershell
$postcode = New-WebServiceProxy -URI "http://www.webservicex.net/uklocation.asmx?wsdl"
$postcode.GetUKLocationByPostcode('SW1A')
```

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

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

```powershell
$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.

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**

Checkpoin-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.

```powershell
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.

```powershell
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.

```powershell
$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.

```powershell
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.

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.

```powershell
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.

<table>
<thead>
<tr>
<th>Status</th>
<th>Name</th>
<th>DisplayName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>IapiSrv</td>
<td>Telephony</td>
</tr>
<tr>
<td>Running</td>
<td>TermService</td>
<td>Terminal Services</td>
</tr>
<tr>
<td>Stopped</td>
<td>ThreadORDER</td>
<td>Thread Ordering Server</td>
</tr>
<tr>
<td>Running</td>
<td>TrkMs</td>
<td>Distributed Link Tracking Client</td>
</tr>
<tr>
<td>Stopped</td>
<td>TrustedInstaller</td>
<td>Windows Modules Installer</td>
</tr>
</tbody>
</table>

**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:\>

![Administrator: Windows PowerShell](image)

[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.

```powershell
$sessions = New-PSSession -ComputerName Test01
Invoke-Command -Session $sessions -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.

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.

```powershell
$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

<table>
<thead>
<tr>
<th>Administrator: Windows PowerShell</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS C:&gt; New-PSSessionOption</td>
</tr>
<tr>
<td>MaximumConnectionRedirectionCount : 5</td>
</tr>
<tr>
<td>NoCompression                   : False</td>
</tr>
<tr>
<td>NoMachineProfile                : False</td>
</tr>
<tr>
<td>ProxyAccessType                 : None</td>
</tr>
<tr>
<td>ProxyAuthentication             : Negotiate</td>
</tr>
<tr>
<td>ProxyCredential                 :</td>
</tr>
<tr>
<td>SkipCNCheck                    : False</td>
</tr>
<tr>
<td>SkipCNCheck                    : False</td>
</tr>
<tr>
<td>SkipRevocationCheck             : False</td>
</tr>
<tr>
<td>OperationTimeout                : 00:03:00</td>
</tr>
<tr>
<td>NoEncryption                   : False</td>
</tr>
<tr>
<td>UseUIRef                       : False</td>
</tr>
<tr>
<td>Culture                        :</td>
</tr>
<tr>
<td>MaximumReceivedDataSizePerCommand :</td>
</tr>
<tr>
<td>MaximumReceivedObjectSize       :</td>
</tr>
<tr>
<td>ApplicationArguments            :</td>
</tr>
<tr>
<td>OpenTimeout                    : 00:03:00</td>
</tr>
<tr>
<td>CloseWaitTimeout               : 00:01:00</td>
</tr>
<tr>
<td>IdleTimeout                    : 00:04:00</td>
</tr>
</tbody>
</table>

PS C:\> .

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.
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

```
Get-PSSession

Id Name             ComputerName | State     | ConfigurationName          | Availability
--- ------ ---------- | ---------- | --------------------------- | ----------
  1     Session1   test01        | Opened    | Microsoft.PowerShell       | Available
  2     Session2   test01        | Opened    | Microsoft.PowerShell       | Available
  3     Session3   test01        | Opened    | Microsoft.PowerShell       | Available

PS C:\>  
```

Get session 3.

Get-PSSession -Id 3

```
Get-PSSession -Id 3

Id Name     ComputerName | State     | ConfigurationName          | Availability
--- ------ ---------- | ---------- | --------------------------- | ----------
  3     Session3   test01 | Opened    | Microsoft.PowerShell       | Available

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

```
Get-PSSession -ComputerName Test01

Id Name     ComputerName | State     | ConfigurationName          | Availability
--- ------ ---------- | ---------- | --------------------------- | ----------
  1     Session1   test01 | Opened    | Microsoft.PowerShell       | Available
  2     Session2   test01 | Opened    | Microsoft.PowerShell       | Available
  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**
**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*'}}
Remove-PSSession -Id 1
```

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
```
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

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.

```
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.

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
```

**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`.

```powershell
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.
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**.

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

**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.

```powershell
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.

**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.

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.

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

<table>
<thead>
<tr>
<th>WSMANConfig:</th>
<th>ComputerName</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>test01</td>
<td>Container</td>
</tr>
<tr>
<td></td>
<td>localhost</td>
<td>Container</td>
</tr>
</tbody>
</table>
```

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

![Administrator: Windows PowerShell]

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

<table>
<thead>
<tr>
<th>WSMANConfig:</th>
<th>Microsoft.WSMan.Management\WSMan::\localhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<tr>
<td>MaxEnvelopeSizeKB</td>
<td>150</td>
</tr>
<tr>
<td>MaxTimeouts</td>
<td>60000</td>
</tr>
<tr>
<td>MaxBatchItems</td>
<td>32000</td>
</tr>
<tr>
<td>MaxProviderRequests</td>
<td>4294967295</td>
</tr>
<tr>
<td>Client</td>
<td>Container</td>
</tr>
<tr>
<td>Service</td>
<td>Container</td>
</tr>
<tr>
<td>Shell</td>
<td>Container</td>
</tr>
<tr>
<td>Listener</td>
<td>Container</td>
</tr>
<tr>
<td>Plugin</td>
<td>Container</td>
</tr>
<tr>
<td>ClientCertificate</td>
<td>Container</td>
</tr>
</tbody>
</table>
```

```
cd Shell
dir | Format-Table -AutoSize
```
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-WManCredSSP**
- **Disconnect-WMan**
- **Enable-PSRemoting**
- **Enable-WManCredSSP**
- **Get-WManCredSSP**
- **Get-WManInstance**
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/wsmapidentity.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.

```powershell
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
```

**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.

```bash
Get-WSManInstance wmicimv2/win32_service
    -SelectorSet @{name="BITS"} -ComputerName Test01
```

Notice that you receive many properties for the BITS service.

![BITS service properties](image)

Display management information for the **WS-Management** listener configuration on the remote computer Test01.

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

Notice that you receive a number of properties of the listener.
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`
Creation of the new WSManInstance using HTTPS is as follows:

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

You can verify this remotely using the `Get-WManInstance` cmdlet.

```
Get-WManInstance winrm/config/listener
-SelectorSet @{Address="*";Transport="https"} -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-WMan`
- `Disable-WManCredSSP`
- `Disconnect-WMan`
- `Enable-PSRemoting`
- `Enable-WManCredSSP`
- `Get-WManCredSSP`
- `Get-WManInstance`
- `Invoke-WManAction`
- `New-PSSession`
- `New-WManSessionOption`
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"}
```

![Image of PowerShell window](image)

**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.

**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.

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

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.

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.

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.

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](https://technet.microsoft.com/en-us/library/hh848501.aspx) with the **Connect-WSMan** cmdlet.

**Example:**


**Disconnect-WSMan** -ComputerName Test01

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

Support for the use of [WS-Management](https://technet.microsoft.com/en-us/library/hh848501.aspx) in PowerShell is provided as part of the 2.0 release.

**Related Cmdlets**

- [Connect-WSMan](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Disable-WSManCredSSP](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Enable-PSRemoting](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Enable-WSManCredSSP](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Invoke-WSManAction](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Remove-WSManInstance](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Set-WSManInstance](https://technet.microsoft.com/en-us/library/hh848501.aspx)
- [Set-WSManQuickConfig](https://technet.microsoft.com/en-us/library/hh848501.aspx)
#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.

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

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

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

**Extras:**

Ravikanth Chaganti has an excellent post covering this cmdlet in more detail [here](https://blog.ravikanth.ch/2015/04/09/import-pssession/).

**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.

```powershell
$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.

The contents of the BITSCommands.psd1 file are below:

```powershell
# Explicit creating module
# generated on 05/02/2010 13:33:16
# involved with the following command line: Export-PSSession -Session $session1 -Module BITSTransfer -OutputModule BITSCommands -AllowClobber
#
$PSModuleBase = 'c:\windows\system32\WindowsPowerShell\v1.0\modules\BITSCommands\bin
Description = 'Explicit creating for http://test01/remote'
ModuleVersion = '1.6'
Provider = [System.Collections.ArrayList]@{9}'
```

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](http://www.chaganti.com/...).
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:

```
ID | Script          | Line | Command | Variable | Action
---|-----------------|------|---------|----------|---------
0  | Bowling.ps1     | 3    |         |          |         
```

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.

```
Entering debug mode. Use h or ? for help.
Hit Line breakpoint on 'C:\Bowling.ps1:3'
Bowling.ps1:3 $iPoints = 0
[00:03]: PS C:\>> Exit
```

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

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.

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

Get-PSBreakpoint -Type Variable

Notice only one breakpoint is returned this time.

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.

```
| Column  | : Ø |
| Line    | : 3 |
| Action  |     |
| Enabled | : False |
| HitCount| : Ø |
| Id      | : Ø |
| Script  | : C:\Bowling.psl |
```

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.

![Select Administrator: Windows PowerShell](image)

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.

Check existing breakpoints and remove all of them.

Get-PSBreakpoint
Get-PSBreakpoint | Remove-PSBreakpoint

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:

![Initial history](image1)

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

![Removed commands](image2)

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"

```
- <_Obj Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
- <ObjRefId="0">
  <TN>System.Object</TN>
  <ToString>$MaximumHistoryCount = 1</ToString>
- <Props>
  <S4 N="id">2</S4>
  <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
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.

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

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

![Event Log Table]

You can create entries in this log using the **Write-EventLog** cmdlet, e.g.

```powershell
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.

![Event Viewer Screenshot]

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.

```powershell
[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:**

![Before](image1)

**After:**

![After](image2)

**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.MaxFileSize = 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.

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
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.

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.

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*.

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

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

![Image of command output]

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*.

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

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

![Image of command output]

**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.

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.
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 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.

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-Module 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.

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.

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
```

**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**
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-PSProvider | 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
```

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
```

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

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
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.

```powershell
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**.

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.

```powershell
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**.
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.

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
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 a fewer options to manipulate it with.

**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.

```powershell
Start-Transaction
$ts.Append("PowerShell")
Use-Transaction -TransactedScript {$ts.Append(" Version 2")} -UseTransaction
$ts.ToString()
```
Note that the current value of the string only contains the text 'PowerShell'.

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'.

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.)

**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.

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](https://en.wikipedia.org/wiki/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.

```powershell
$herestring = @'
Fruit1 = Orange
Fruit2 = Apple
'@

$herestring | ConvertFrom-StringData
```

You will notice that the data is now in the form of a hash table

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit2</td>
<td>Apple</td>
</tr>
<tr>
<td>Fruit1</td>
<td>Orange</td>
</tr>
</tbody>
</table>

**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

```powershell
(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")
}
```

147
#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.

```powershell
$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>
    <Property Name="CanPauseAndContinue" Type="System.Boolean">False</Property>
    <Property Name="CanStop" Type="System.Boolean">False</Property>
    <Property Name="DisplayName" Type="System.String">Base Filtering Engine</Property>
    <Property Type="System.ServiceProcess.ServiceController[]">
    </Property>
    <Property Type="System.ServiceProcess.ServiceController[]">
    </Property>
    <Property Type="System.ServiceProcess.ServiceController[]">
    </Property>
    <Property Type="System.ServiceProcess.ServiceController[]">
    </Property>
  </Object>
  <Object Type="System.ServiceProcess.ServiceController">
    <Property Name="MachineName" Type="System.string">.</Property>
    <Property Name="ServiceName" Type="System.String">BFE</Property>
    <Property Name="ServicesDependedOn" Type="System.ServiceProcess.ServiceController[]">
    </Property>
    <Property Name="ServiceHandle" Type="SafeServiceHandle">SafeServiceHandle</Property>
    <Property Name="Status" Type="System.ServiceProcess.ServiceControllerStatus">Running</Property>
    <Property Name="ServiceType" Type="System.ServiceProcess.ServiceControllerServiceType">Win32Service</Property>
    <Property Name="Site" Type="System.ComponentModel.Model.ISite" />
    <Property Name="Container" Type="System.ComponentModel.Model.IContainer" />
  </Object>
</Objects>
```

**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.

```powershell
$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.

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**.
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.

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.

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**.

```powershell
Set-WmiInstance -Class Win32_WMISetting
-Argument @{MaxLogFileSize=13112}
```

You will notice that the **MaxLogFileSize** value is updated:

![PowerShell Output](image.png)

**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.

```powershell
$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'" -SourceIdentifier "Check for New Processes" -Action "A new process started at " + (Get-Date) | Out-File c:\log.txt -Append
```

After running the above command and then starting a process the below is automatically written to `c:\log.txt` after a few seconds.

```
A new process started at 04/20/2010 13:28:12
```

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
```

159
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'"
$query
Register-ObjectEvent -InputObject $processWatcher -EventName "EventArrived" -Action { "A new process started at " + (Get-Date) | Out-File c:\log.txt -Append}
```

After running the above commands and then starting a process the below is automatically written to c:\log.txt after a few seconds.

```
A new process started at 04/21/2010 14:29:05
```

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
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.

```powershell
$query = New-Object System.Management.WqlEventQuery
"__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa 'Win32_Process"
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

```
SubscriptionId : 1
EventName      : EventArrived
SourceIdentifier: 664ff1f-29ea-49a1-ac1b-10f743c02e2
SupportEvent   : False
ForwardEvent   : False
```

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
}
```

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
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.

```powershell
function Enable-ProcessCreationEvent
{
    $query = New-Object System.Management.WqlEventQuery
        "__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa 'Win32_Process'"
    $query
    $identifier = "WMI.ProcessCreated"

    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
    $identifier -Action {
        $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.

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
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.

```powershell
function Enable-ProcessCreationEvent {
    $query = New-Object System.Management.WqlEventQuery
       "__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa 'Win32_Process'

    $query
    $identifier = "WMI.ProcessCreated"

    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
    $identifier -Action {
        [void] (New-Event -SourceIdentifier "PowerShell.ProcessCreated" -
        Sender $args[0] -EventArguments
        $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. Get-Event retrieves details of this event.
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.

```powershell
function Enable-ProcessCreationEvent
{
    $query = New-Object System.Management.WqlEventQuery
        "__InstanceCreationEvent", (New-Object TimeSpan 0,0,1), "TargetInstance isa
        'Win32_Process'

    $query
    $identifier = "WMI.ProcessCreated"
    Register-ObjectEvent $processWatcher "EventArrived" -SupportEvent
    $identifier -Action { [void] (New-Event -SourceIdentifier "PowerShell.ProcessCreated" -
        Sender $args[0] -EventArguments $args[1].SourceEventArgs.NewEvent.TargetInstance) }
}
Wait-Event -SourceIdentifier PowerShell.ProcessCreated
```

Before opening Windows Calculator:

![Administrator: Windows PowerShell](Image)

You will notice that after the opening of Windows Calculator the event is triggered and the prompt is returned to the user.
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.

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.

```powershell
Get-Event
Remove-Event -SourceIdentifier Timer
Get-Event
```

You will see that the event has been cleared.

![PowerShell output image]

**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.

```
Get-Process -Name 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.

```
Get-Process -Name Notepad
Wait-Process -Name Notepad
```

**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.

```
Get-Process -Name 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.

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.
Now check the impact on the PSSessionconfiguration - the **AccessDenied** permission has been applied to **Everyone**.

![PowerShell screenshot](image)

**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**
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}
```

You will see it is possible to work through the debug info to find out what is happening:

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.
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](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_version_parameters) 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.
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.

```powershell
$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.

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 $PSUCulture 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 $PSUCulture 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.

```powershell
$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!)

```powershell
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#.

**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
@csharp
public class TakeAway {
    public static int Minus(int a, int b)
    {
        return (a - b);
    }
}
@csharp
Add-Type -TypeDefinition $csharp
[TakeAway]::Minus(10,7)
```

You will see that we get the expected answer of 3:

**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