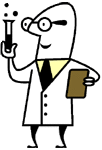
**PowerShell Workflows: The Basics**



Dr

December 26th, 2012

**Summary:** Windows PowerShell MVP Richard Siddaway talks about the basics of Windows PowerShell 3.0 workflows.  
Microsoft Scripting Guy, Ed Wilson, is here. Today, we are fortunate to have a guest blog post by Windows PowerShell MVP Richard Siddaway. Richard has written a [number of guest Hey, Scripting Guy! Blog posts](http://blogs.technet.com/b/heyscriptingguy/archive/tags/richard+siddaway/), and he has also written two books on Windows PowerShell. His most recent book, [PowerShell in Depth](http://www.manning.com/jones2/), is co-written with fellow MVPs Don Jones and Jeffrey Hicks.  
Now, take it away, Richard …

**PowerShell Workflows: The Basics**

Thanks, Ed.  
Now, if you’ve read anything I’ve written over the last six years, you’ll know that I think that Windows PowerShell is the best thing to ever occur in the Windows eco-system. With Windows PowerShell 3.0, it just got better. WMI has been one of my main interests for the last few years, but one feature in Windows PowerShell 3.0 keeps dragging me away—that feature is Windows PowerShell workflows.  
Whenever I come across a new piece of technology I want to know three things:

1. What it is and how it works?
2. Why it’s important?
3. What can it do for me?

This series of articles will explain the basics of workflows, how they differ from the Windows PowerShell you are used to, and how to get the best out of them. By the end of the series, you will be able to answer the questions I posed and see where workflows fit into your automation and integration strategy.  
One definition of a workflow is that it is a sequence of automated steps or activities that can execute tasks, or retrieve data, on local or remote machines.  
That may sound like a Windows PowerShell function or script, and you wouldn’t be completely wrong for thinking that, but—and it’s a big “but”—Windows PowerShell workflows are designed for scenarios where these attributes are required:

* Long-running activities.
* Repeatable activities.
* Frequently executed activities.
* Running activities in parallel across one or more machines.
* Interruptible activities that can be stopped and re-started, which includes surviving a reboot of the system against which the workflow is executing.

OK, so I’ve sold you on the idea that workflows are a good thing—but what do they look like? There is a tradition in “programming” circles that any new language is introduced through a “Hello world” program. Here is the world’s simplest workflow that is just that.

workflow helloworld {

“Hello World”

}  
Running this code creates the workflow but it doesn’t run it—much the same as a Windows PowerShell function. To execute a workflow, simply type its name at the prompt and press Enter. The first time you run an individual workflow, be prepared for a slight delay before it actually executes.  
Our workflow may look like a function with a different keyword but don’t be fooled—the most important point to understand about workflows is that they look like Windows PowerShell but they aren’t. I’ll repeat this statement as we explore workflows, and by the end of the articles, you’ll understand those differences.  
You’ve seen that Windows PowerShell 3.0 has introduced the workflow keyword. There are a number of workflow-related key words that you need to be aware of:

* Workflow
* Parallel
* Foreach –parallel
* Sequence
* InlineScript
* Checkpoint-workflow
* Suspend-workflow

One of the key features for workflows is that they can execute commands in parallel. Consider this set of Windows PowerShell commands:

Get-CimInstance –ClassName Win32\_OperatingSystem

Get-Process –Name PowerShell\*

Get-CimInstance –ClassName Win32\_ComputerSystem

Get-Service –Name s\*  
 In what order would you expect the results to be returned if you run these commands in a script or Windows PowerShell function?  
Assuming that there were no problems, I would expect to see:

1. The Win32\_OperatingSystem data
2. The Windows PowerShell process data
3. The Win32\_ComputerSystem data
4. The data for services beginning with the letter “s”

Now let’s turn this into a parallelized workflow.

workflow paralleltest {

 parallel {

   Get-CimInstance –ClassName Win32\_OperatingSystem

   Get-Process –Name PowerShell\*

   Get-CimInstance –ClassName Win32\_ComputerSystem

   Get-Service –Name s\*

  }

}  
If you want a set of commands to execute in parallel, all you need to do is add the parallel keyword and the code between the brackets {} will be executed in parallel.  
In which order do you think the data will be returned?  
You can’t tell! You can run this workflow a number of times and the data may be returned in a different order each time you run it!  
*If you are running workflow activities in parallel there are no guarantees as to the order in which data will be returned. You cannot assume that one piece of data will be returned before another.*  
I’ve used the phrase “workflow activities” a number of times—quite deliberately. Remember that I said a workflow looks like Windows PowerShell but isn’t. You may think that the code in the paralleltest workflow is Windows PowerShell but it isn’t.  
The Windows PowerShell workflow functionality is built on the .NET Framework Windows Workflow Foundation (WWF). Most, but not all, Windows PowerShell cmdlets have been mapped to workflow activities (I’ll cover the ones that haven’t been mapped later). That is what you were running—workflow activities. Your Windows PowerShell workflow code is translated to XAML for WWF to run. You can even import the XAML into Visual Studio if required.   
Try this to view the XAML code:

Get-command paralletest | format-list \*  
 For the most part, the syntax for using a workflow activity is the same as using a cmdlet but there are a few differences—the most obvious of which is the **–ComputerName** parameter. In workflow activities, this is replaced by a **–PSComputerName** parameter. We can illustrate this with another workflow.

workflow foreachptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

    Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

   }

}  
You can run the workflow like this:

foreachptest  -Computers “server01”, “server02”, “server03”   
Again, there is no guarantee what order the data will be returned.  
What about the situations where commands have to be executed in a certain order? That functionality is provided by the sequence key word. Anything in a sequence{} block is run in the order you type it. You could modify the last workflow like this:

workflow foreachpstest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

    sequence {

      Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

      Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

    }

   }

}

foreachpstest  -Computers “server01”, “server02”, “server03”   
 In this workflow, you will access the three computers in parallel but for each computer the commands …  
      Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer  
      Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer  
 … will be run in that order—and always in that order.  
Confused yet? No? That’s good, so let’s try this.

workflow foreachpsptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

    sequence {

      Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

      Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

      $disks = Get-WmiObject -Class Win32\_LogicalDisk `

         -Filter “DriveType = 3” –PSComputerName $computer

      foreach -parallel ($disk in $disks){

        sequence {

          $vol = Get-WmiObject -Class Win32\_Volume `

          -Filter “DriveLetter = ‘$($disk.DeviceID)'” `

          –PSComputerName $computer

          Invoke-WmiMethod -Path $($vol.\_\_PATH) -Name DefragAnalysis

        }

      }

    }

   }

}

foreachpstest  -Computers “server01”, “server02”, “server03”   
The workflow takes a list of computer names and in parallel performs a number of actions. Those actions occur in the order given because we are using the sequence keyword. The **Win32\_ComputerSystem** and **Win32\_OperatingSystem** instances are straightforward because you only get one of each. The workflow then gets the logical disks, BUT because there are a number of them, we want to process them in parallel so we use:

foreach -parallel ($disk in $disks)  
Each disk has the equivalent **Win32\_Volume** class accessed and the **\_\_PATH** property is used to invoke the **DefragAnalysis** method.  
Normally, you would expect to be able to do this:

Get-WmiObject -Class Win32\_Volume -Filter “DriveLetter = ‘C:'” |

Invoke-WmiMethod -Name DefragAnalysis  
But workflows work across WSMAN (like remoting), and you get deserialized objects returned so you can’t pipe. You can get around it in this case by using the **WMI \_\_PATH** property of the volume.   
The other thing that I hope you noticed is that I’m always using the parameter names in the workflow activities (cmdlets). This is because workflows don’t allow the use of position parameters. All those neat one-liners you’ve built up—can’t use ‘em. You have to use the parameter name!  
One of the problems that people sometimes have with WMI is that the default formatting doesn’t display all of the data. One standard approach to overcome this issue is to use Format-List \*. So you might be tempted to do this:

workflow foreachptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

    Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer |

    Format-List

   }

}   
Unfortunately, it won’t work and you will see an error message like this:

At line:5 char:5

+     Format-List

+     ~~~~~~~~~~~

Cannot call the ‘Format-List’ command. Other commands from this module have been packaged as workflow activities, but this command was specifically excluded. This is likely because the command requires an interactive Windows PowerShell session, or has behavior not suited for workflows. To run this command anyway, place it within an inline-script (InlineScript {Format-List }) where it will be invoked in isolation.

    + CategoryInfo          : ParserError: (:) [], ParseException

    + FullyQualifiedErrorId : CommandActivityExcluded  
**Format-List** is one of the cmdlets that haven’t been turned into workflow activities as the error message explains. It goes on to recommend using an **InlineScript**, which is effectively a Windows PowerShell script block inside your workflow. You can use it like this:

workflow foreachpitest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

     InlineScript {

       Get-WmiObject –Class Win32\_OperatingSystem –ComputerName $using:computer |

       Format-List

     }

   }

}   
Two things to notice with this workflow. First, it reverts to using the **–ComputerName** parameter. This is because we are dealing with the cmdlet not the workflow activity. Second, the way the **$computer** variable is accessed—it has to be **$using:computer**. This is due to the way workflow scopes work. I’ll explain it fully later, but for now, if you want to use a variable that has been defined higher in the workflow in an **InlineScriptBlock**, then you have to use the **$using** modifier.  
Workflows—love ‘em or hate ‘em—they are here to stay. In this opening article, you have seen that workflows look like Windows PowerShell, but have a number of differences. You have also met some of the workflow keywords:

* Parallel
* Foreach –parallel
* Sequence
* InlineScript

Next time, we’ll look at some of the restrictions that workflows have and how you can overcome them.  
~Richard  
Thank you, Richard. I look forward to seeing more from you on Windows PowerShell workflow. Join us tomorrow when we will have a guest blog post written by Microsoft PFE Jason Walker.  
I invite you to follow me on [Twitter](http://bit.ly/scriptingguystwitter) and [Facebook](http://bit.ly/scriptingguysfacebook). If you have any questions, send email to me at [scripter@microsoft.com](http://blogs.technet.commailto:scripter@microsoft.com), or post your questions on the [Official Scripting Guys Forum](http://bit.ly/scriptingforum). See you tomorrow. Until then, peace.

**Ed Wilson, Microsoft Scripting Guy**