Windows PowerShell: The Advanced Function Lifecycle

The advanced functions of Windows PowerShell—called Script cmdlets—can be somewhat confusing, but here's a way to direct their setup and cleanup functions.

Don Jones

There has been an ongoing point of confusion for a number of students in some of my on-site Windows PowerShell classes. I hope that exploring it in more detail here will help unwind some confusion for you as well. The topic is the advanced functions of Windows PowerShell, informally known as script cmdlets. The template for this type of function looks like this:

```
Function Do-Something {
        [CmdletBinding()]
        param(
                [Parameter(Mandatory=$True,
                ValueFromPipeline=$True)]
                [string[]]$computername
        )
        BEGIN {}
        PROCESS {}
        END {}
}
```

There are a couple of confusing aspects to these cmdlets. For example, in this one I've defined an input parameter, -computername. This can accept input from the pipeline. This means you can call this function in two distinct ways. First, you can have strings piped into it -- such as from a text file that contains one computer name per line:

```
Get-Content names.txt | Do-Something
```

You can also have one or more computer names passed directly to the parameter, without using the pipeline at all:

```
Do-Something –computername SERVER1,SERVER2
```

In the first example, the function's BEGIN block executes first. Then, the PROCESS block executes once for each piped-in computer name. The $computername variable contains only one computer name at a time. Finally, once they're all processed, the END block executes once.

The same lifecycle occurs when you call the function and don't pipe in anything, but instead provide values entirely to the parameters. It's easy to do set-up and clean-up tasks that run in both situations.

However, dealing with $computername parameters can be confusing. In the first example, it will only contain one value at a time. In the second, it will contain one or more values, depending on what is assigned to the –computername parameter. Simply putting a ForEach loop within the PROCESS block is a good way to deal with the $computername issue:

```
Function Do-Something {
        [CmdletBinding()]
        param(
                [Parameter(Mandatory=$True,
                ValueFromPipeline=$True)]
                [string[]]$computername
        )
        BEGIN {}
        PROCESS {
                Foreach ($computer in $computername) {
                        # use $computer here
                }
}
        END {}
}
```

With this technique, the $computer variable will be guaranteed to contain just one computer name at a time, so I can work with that instead of $computername.

The set-up and clean-up is a bit trickier. You don't want to do that setup directly within the PROCESS block, since that block can run multiple times when objects are piped-in. Instead, you can put setup directly into the BEGIN block. There are many ways you could choose to deal with this, but here's how I do it:

```
Function Do-Something {
        [CmdletBinding()]
        param(
                [Parameter(Mandatory=$True,
                ValueFromPipeline=$True)]
                [string[]]$computername
        )
        BEGIN {
# Do Setup Here
}
        PROCESS {
                Foreach ($computer in $computername) {
                        # use $computer here
                }
    }
    END {}
}
```

Performing the same trick for the clean-up tasks is no more difficult. Just put the code into an END block.

There are some sneaky ways to solve the problem, but I often tend to take a simpler approach. When I need some kind of clean-up activity, such as closing a database connection, I'll just forget about using the BEGIN and END blocks altogether. I'll have my PROCESS block assume it's only going to run once, and just put all set-up and clean-up within that block.

It can be a bit wasteful. I may be opening and closing a database connection repeatedly, for example, but it's effective. It eliminates the need for me to have difficult-to-follow perambulations in the code.

Programming these functions to handle both pipeline and non-pipeline scenarios can be tricky. In future articles, I'll share some of the ways other folks have solved this problem, so you have options for use in your own environment.

Don Jones *is a Microsoft MVP Award recipient and author of "Learn Windows PowerShell in a Month of Lunches" (Manning Publications, 2010), a book designed to help any administrator become effective with Windows PowerShell. Jones also offers public and on-site Windows PowerShell training. Contact him through his Web site at ConcentratedTech.com.*

Related Content

- Windows PowerShell: Package and Distribute Custom Windows PowerShell Tools
- Windows PowerShell: Think Commands, Not Scripts
- Windows PowerShell: Make a Command into a Reusable Tool

© 2016 Microsoft