**Creating .NET and COM Objects (New-Object)**

* Article
* 10/12/2021
* 10 minutes to read
* 3 contributors

There are software components with .NET Framework and COM interfaces that enable you to perform many system administration tasks. Windows PowerShell lets you use these components, so you are not limited to the tasks that can be performed by using cmdlets. Many of the cmdlets in the initial release of Windows PowerShell do not work against remote computers. We will demonstrate how to get around this limitation when managing event logs by using the .NET Framework **System.Diagnostics.EventLog** class directly from Windows PowerShell.

**Using New-Object for Event Log Access**

The .NET Framework Class Library includes a class named **System.Diagnostics.EventLog** that can be used to manage event logs. You can create a new instance of a .NET Framework class by using the New-Object cmdlet with the **TypeName** parameter. For example, the following command creates an event log reference:

PowerShellCopy

New-Object -TypeName System.Diagnostics.EventLog

OutputCopy

Max(K) Retain OverflowAction Entries Name

------ ------ -------------- ------- ----

Although the command has created an instance of the EventLog class, the instance does not include any data. That is because we did not specify a particular event log. How do you get a real event log?

**Using Constructors with New-Object**

To refer to a specific event log, you need to specify the name of the log. New-Object has an **ArgumentList** parameter. The arguments you pass as values to this parameter are used by a special startup method of the object. The method is called a **constructor** because it is used to construct the object. For example, to get a reference to the Application log, you specify the string 'Application' as an argument:

PowerShellCopy

New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application

OutputCopy

Max(K) Retain OverflowAction Entries Name

------ ------ -------------- ------- ----

16,384 7 OverwriteOlder 2,160 Application

**Note**

Since most of the .NET Framework core classes are contained in the System namespace, Windows PowerShell will automatically attempt to find classes you specify in the System namespace if it cannot find a match for the typename you specify. This means that you can specify Diagnostics.EventLog instead of System.Diagnostics.EventLog.

**Storing Objects in Variables**

You might want to store a reference to an object, so you can use it in the current shell. Although Windows PowerShell lets you do a lot of work with pipelines, lessening the need for variables, sometimes storing references to objects in variables makes it more convenient to manipulate those objects.

Windows PowerShell lets you create variables that are essentially named objects. The output from any valid Windows PowerShell command can be stored in a variable. Variable names always begin with $. If you want to store the Application log reference in a variable named $AppLog, type the name of the variable, followed by an equal sign and then type the command used to create the Application log object:

PowerShellCopy

$AppLog = New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application

If you then type $AppLog, you can see that it contains the Application log:

PowerShellCopy

$AppLog

OutputCopy

Max(K) Retain OverflowAction Entries Name

------ ------ -------------- ------- ----

16,384 7 OverwriteOlder 2,160 Application

**Accessing a Remote Event Log with New-Object**

The commands used in the preceding section target the local computer; the Get-EventLog cmdlet can do that. To access the Application log on a remote computer, you must supply both the log name and a computer name (or IP address) as arguments.

PowerShellCopy

$RemoteAppLog = New-Object -TypeName System.Diagnostics.EventLog Application,192.168.1.81

$RemoteAppLog

OutputCopy

Max(K) Retain OverflowAction Entries Name

------ ------ -------------- ------- ----

512 7 OverwriteOlder 262 Application

Now that we have a reference to an event log stored in the $RemoteAppLog variable, what tasks can we perform on it?

**Clearing an Event Log with Object Methods**

Objects often have methods that can be called to perform tasks. You can use Get-Member to display the methods associated with an object. The following command and selected output show some the methods of the EventLog class:

PowerShellCopy

$RemoteAppLog | Get-Member -MemberType Method

OutputCopy

TypeName: System.Diagnostics.EventLog

Name MemberType Definition

---- ---------- ----------

...

Clear Method System.Void Clear()

Close Method System.Void Close()

...

GetType Method System.Type GetType()

...

ModifyOverflowPolicy Method System.Void ModifyOverflowPolicy(Overfl...

RegisterDisplayName Method System.Void RegisterDisplayName(String ...

...

ToString Method System.String ToString()

WriteEntry Method System.Void WriteEntry(String message),...

WriteEvent Method System.Void WriteEvent(EventInstance in...

The 'Clear()` method can be used to clear the event log. When calling a method, you must always follow the method name by parentheses, even if the method does not require arguments. This lets Windows PowerShell distinguish between the method and a potential property with the same name. Type the following to call the **Clear** method:

PowerShellCopy

$RemoteAppLog.Clear()

Type the following to display the log. You will see that the event log was cleared, and now has 0 entries instead of 262:

PowerShellCopy

$RemoteAppLog

```Output

Max(K) Retain OverflowAction Entries Name

------ ------ -------------- ------- ----

512 7 OverwriteOlder 0 Application

**Creating COM Objects with New-Object**

You can use New-Object to work with Component Object Model (COM) components. Components range from the various libraries included with Windows Script Host (WSH) to ActiveX applications such as Internet Explorer that are installed on most systems.

New-Object uses .NET Framework Runtime-Callable Wrappers to create COM objects, so it has the same limitations that .NET Framework does when calling COM objects. To create a COM object, you need to specify the **ComObject** parameter with the Programmatic Identifier or **ProgId** of the COM class you want to use. A complete discussion of the limitations of COM use and determining what ProgIds are available on a system is beyond the scope of this user's guide, but most well-known objects from environments such as WSH can be used within Windows PowerShell.

You can create the WSH objects by specifying these progids: **WScript.Shell**, **WScript.Network**, **Scripting.Dictionary**, and **Scripting.FileSystemObject**. The following commands create these objects:

PowerShellCopy

New-Object -ComObject WScript.Shell

New-Object -ComObject WScript.Network

New-Object -ComObject Scripting.Dictionary

New-Object -ComObject Scripting.FileSystemObject

Although most of the functionality of these classes is made available in other ways in Windows PowerShell, a few tasks such as shortcut creation are still easier to do using the WSH classes.

**Creating a Desktop Shortcut with WScript.Shell**

One task that can be performed quickly with a COM object is creating a shortcut. Suppose you want to create a shortcut on your desktop that links to the home folder for Windows PowerShell. You first need to create a reference to **WScript.Shell**, which we will store in a variable named **$WshShell**:

PowerShellCopy

$WshShell = New-Object -ComObject WScript.Shell

Get-Member works with COM objects, so you can explore the members of the object by typing:

PowerShellCopy

$WshShell | Get-Member

OutputCopy

TypeName: System.\_\_ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name MemberType Definition

---- ---------- ----------

AppActivate Method bool AppActivate (Variant, Va...

CreateShortcut Method IDispatch CreateShortcut (str...

...

Get-Member has an optional **InputObject** parameter you can use instead of piping to provide input to Get-Member. You would get the same output as shown above if you instead used the command **Get-Member -InputObject $WshShell**. If you use **InputObject**, it treats its argument as a single item. This means that if you have several objects in a variable, Get-Member treats them as an array of objects. For example:

PowerShellCopy

$a = 1,2,"three"

Get-Member -InputObject $a

OutputCopy

TypeName: System.Object[]

Name MemberType Definition

---- ---------- ----------

Count AliasProperty Count = Length

...

The **WScript.Shell CreateShortcut** method accepts a single argument, the path to the shortcut file to create. We could type in the full path to the desktop, but there is an easier way. The desktop is normally represented by a folder named Desktop inside the home folder of the current user. Windows PowerShell has a variable $Home that contains the path to this folder. We can specify the path to the home folder by using this variable, and then add the name of the Desktop folder and the name for the shortcut to create by typing:

PowerShellCopy

$lnk = $WshShell.CreateShortcut("$Home\Desktop\PSHome.lnk")

When you use something that looks like a variable name inside double-quotes, Windows PowerShell tries to substitute a matching value. If you use single-quotes, Windows PowerShell does not try to substitute the variable value. For example, try typing the following commands:

PowerShellCopy

"$Home\Desktop\PSHome.lnk"

OutputCopy

C:\Documents and Settings\aka\Desktop\PSHome.lnk

PowerShellCopy

'$Home\Desktop\PSHome.lnk'

OutputCopy

$Home\Desktop\PSHome.lnk

We now have a variable named $lnk that contains a new shortcut reference. If you want to see its members, you can pipe it to Get-Member. The output below shows the members we need to use to finish creating our shortcut:

PowerShellCopy

$lnk | Get-Member

OutputCopy

TypeName: System.\_\_ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}

Name MemberType Definition

---- ---------- ----------

...

Save Method void Save ()

...

TargetPath Property string TargetPath () {get} {set}

We need to specify the **TargetPath**, which is the application folder for Windows PowerShell, and then save the shortcut \*\*$lnk\*\* by calling the **Save** method. The Windows PowerShell application folder path is stored in the variable $PSHome, so we can do this by typing:

PowerShellCopy

$lnk.TargetPath = $PSHome

$lnk.Save()

**Using Internet Explorer from Windows PowerShell**

Many applications (including the Microsoft Office family of applications and Internet Explorer) can be automated by using COM. Internet Explorer illustrates some of the typical techniques and issues involved in working with COM-based applications.

You create an Internet Explorer instance by specifying the Internet Explorer ProgId, **InternetExplorer.Application**:

PowerShellCopy

$ie = New-Object -ComObject InternetExplorer.Application

This command starts Internet Explorer, but does not make it visible. If you type Get-Process, you can see that a process named iexplore is running. In fact, if you exit Windows PowerShell, the process will continue to run. You must reboot the computer or use a tool like Task Manager to end the iexplore process.

**Note**

COM objects that start as separate processes, commonly called *ActiveX executables*, may or may not display a user interface window when they start up. If they create a window but do not make it visible, like Internet Explorer, the focus will generally move to the Windows desktop and you must make the window visible to interact with it.

By typing $ie | Get-Member, you can view properties and methods for Internet Explorer. To see the Internet Explorer window, set the Visible property to $true by typing:

PowerShellCopy

$ie.Visible = $true

You can then navigate to a specific Web address by using the Navigate method:

PowerShellCopy

$ie.Navigate("https://devblogs.microsoft.com/scripting/")

Using other members of the Internet Explorer object model, it is possible to retrieve text content from the Web page. The following command will display the HTML text in the body of the current Web page:

PowerShellCopy

$ie.Document.Body.InnerText

To close Internet Explorer from within PowerShell, call its Quit() method:

PowerShellCopy

$ie.Quit()

This will force it to close. The $ie variable no longer contains a valid reference even though it still appears to be a COM object. If you attempt to use it, you will get an automation error:

PowerShellCopy

$ie | Get-Member

OutputCopy

Get-Member : Exception retrieving the string representation for property "Appli

cation" : "The object invoked has disconnected from its clients. (Exception fro

m HRESULT: 0x80010108 (RPC\_E\_DISCONNECTED))"

At line:1 char:16

+ $ie | Get-Member <<<<

You can either remove the remaining reference with a command like $ie = $null, or completely remove the variable by typing:

PowerShellCopy

Remove-Variable ie

**Note**

There is no common standard for whether ActiveX executables exit or continue to run when you remove a reference to one. Depending on circumstances such as whether the application is visible, whether an edited document is running in it, and even whether Windows PowerShell is still running, the application may or may not exit. For this reason, you should test termination behavior for each ActiveX executable you want to use in Windows PowerShell.

**Getting Warnings About .NET Framework-Wrapped COM Objects**

In some cases, a COM object might have an associated .NET Framework **Runtime-Callable Wrapper** or RCW, and this will be used by New-Object. Since the behavior of the RCW may be different from the behavior of the normal COM object, New-Object has a **Strict** parameter to warn you about RCW access. If you specify the **Strict** parameter and then create a COM object that uses an RCW, you get a warning message:

PowerShellCopy

$xl = New-Object -ComObject Excel.Application -Strict

OutputCopy

New-Object : The object written to the pipeline is an instance of the type "Mic

rosoft.Office.Interop.Excel.ApplicationClass" from the component's primary inte rop assembly. If

this type exposes different members than the IDispatch members , scripts written to work with this

object might not work if the primary intero p assembly is not installed. At line:1 char:17 + $xl =

New-Object <<<< -ComObject Excel.Application -Strict

Although the object is still created, you are warned that it is not a standard COM object.