Microsoft | TechNet

Add-Type

Updated: April 21, 2010

Applies To: Windows PowerShell 2.0

Adds a Microsoft .NET Framework type (a class) to a Windows PowerShell session.

Syntax

```
Add-Type -AssemblyName <string[]> [-IgnoreWarnings] [-PassThru] [<CommonParameters>]

Add-Type [-Name] <string> [-MemberDefinition] <string[]> [-CodeDomProvider <CodeDomProvider>]
[-CompilerParameters <CompilerParameters>] [-Language {<CSharp> | <CSharpVersion3> | <VisualBa
sic> | <JScript>}] [-Namespace <string>] [-OutputAssembly <string>] [-OutputType <OutputAssemb
lyType>] [-ReferencedAssemblies <string[]>] [-UsingNamespace <string[]>] [-IgnoreWarnings] [-P
assThru] [<CommonParameters>]

Add-Type [-Path] <string[]> [-CompilerParameters <CompilerParameters>] [-OutputAssembly <strin
g>] [-OutputType <OutputAssemblyType>] [-ReferencedAssemblies <string[]>] [-IgnoreWarnings] [-
PassThru] [<CommonParameters>]

Add-Type [-TypeDefinition] <string> [-CodeDomProvider <CodeDomProvider>] [-CompilerParameters
<CompilerParameters>] [-Language {<CSharp> | <CSharpVersion3> | <VisualBasic> | <JScript>}] [-
OutputAssembly <string>] [-OutputType <OutputAssemblyType>] [-ReferencedAssemblies <string[]>]
 [-IgnoreWarnings] [-PassThru] [<CommonParameters>]
```

Description

The Add-Type cmdlet lets you define a .NET Framework class in your Windows PowerShell session. You can then instantiate objects (by using the New-Object[1] cmdlet) and use the objects, just as you would use any .NET Framework object. If you add an Add-Type command to your Windows PowerShell profile, the class will be available in all Windows PowerShell sessions.

You can specify the type by specifying an existing assembly or source code files, or you can specify the source code inline or saved in a variable. You can even specify only a method and Add-Type will define and generate the class. You can use this feature to make Platform Invoke (P/Invoke) calls to unmanaged functions in Windows PowerShell. If you specify source code, Add-Type compiles the specified source code and generates an in-memory assembly that contains the new .NET Framework types.

You can use the parameters of Add-Type to specify an alternate language and compiler (CSharp is the default), compiler options, assembly dependencies, the class namespace, the names of the type, and the resulting assembly.

Parameters

-AssemblyName <string[]>

Specifies the name of an assembly that includes the types. Add-Type takes the types from the specified assembly. This parameter is required when you are creating types based on an assembly name.

Enter the full or simple name (also known as the "partial name") of an assembly. Wildcard characters are permitted in the assembly name. If you enter a simple or partial name, Add-Type resolves it to the full name, and then uses the full name to load the assembly.

This parameter does not accept a path or file name. To enter the path to the assembly dynamic-link library (DLL) file, use the Path parameter.

Required?	true
Position?	named
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	true

-CodeDomProvider <CodeDomProvider>

Specifies a code generator or compiler. Add-Type uses the specified compiler to compile the source code. The default is the CSharp compiler. Use this parameter if you are using a language that cannot be specified by using the Language parameter. The CodeDomProvider that you specify must be able to generate assemblies from source code.

Required?	false
Position?	named
Default Value	CSharp compiler
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-CompilerParameters <CompilerParameters>

Specifies the options for the source code compiler. These options are sent to the compiler without revision.

This parameter allows you to direct the compiler to generate an executable file, embed resources, or set command-line options, such as the "/unsafe" option. This parameter implements the CompilerParameters class (System.CodeDom.Compiler.CompilerParameters).

Required?	false

Position?	named
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-IgnoreWarnings

Ignores compiler warnings. Use this parameter to prevent Add-Type from handling compiler warnings as errors.

Required?	false
Position?	named
Default Value	False
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-Language <Language>

Specifies the language used in the source code. Add-Type uses the language to select the correct code compiler.

Valid values are "CSharp", "CSharpVersion3", "VisualBasic", and "JScript". "CSharp" is the default.

Required?	false
Position?	named
Default Value	CSharp
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-MemberDefinition <string[]>

Specifies new properties or methods for the class. Add-Type generates the template code that is

required to support the properties or methods.

You can use this feature to make Platform Invoke (P/Invoke) calls to unmanaged functions in Windows PowerShell. For more information, see the examples.

Required?	true
Position?	2
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-Name <string>

Specifies the name of the class to create. This parameter is required when generating a type from a member definition.

The type name and namespace must be unique within a session. You cannot unload a type or change it. If you need to change the code for a type, you must change the name or start a new Windows PowerShell session. Otherwise, the command fails.

Required?	true
Position?	1
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-Namespace <string>

Specifies a namespace for the type.

If this parameter is not included in the command, the type is created in the Microsoft.PowerShell.Commands.AddType.AutoGeneratedTypes namespace. If the parameter is included in the command with an empty string value or a value of $null, the type is generated in the global namespace.

Required?	false

Position?	named
Default Value	Microsoft.PowerShell.Commands.AddType.AutoGeneratedTypes
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-OutputAssembly <string>

Generates a DLL file for the assembly with the specified name in the location. Enter a path (optional) and file name. Wildcard characters are permitted. By default, Add-Type generates the assembly only in memory.

Required?	false
Position?	named
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	true

-OutputType <OutputAssemblyType>

Specifies the output type of the output assembly. Valid values are Library, ConsoleApplication, and WindowsApplication.

By default, no output type is specified.

This parameter is valid only when an output assembly is specified in the command.

Required?	false
Position?	named
Default Value	None
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-PassThru

Returns a System.Runtime object that represents the types that were added. By default, this cmdlet does not generate any output.

Required?	false
Position?	named
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-Path <string[]>

Specifies the path to source code files or assembly DLL files that contain the types.

If you submit source code files, Add-Type compiles the code in the files and creates an in-memory assembly of the types. The file name extension specified in the value of Path determines the compiler that Add-Type uses.

If you submit an assembly file, Add-Type takes the types from the assembly. To specify an in-memory assembly or the global assembly cache, use the AssemblyName parameter.

Required?	true
Position?	1
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-ReferencedAssemblies <string[]>

Specifies the assemblies upon which the type depends. By default, Add-Type references System.dll and System.Management.Automation.dll. The assemblies that you specify by using this parameter are referenced in addition to the default assemblies.

Required?	false

Position?	named
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-TypeDefinition <string>

Specifies the source code that contains the type definitions. Enter the source code in a string or here-string, or enter a variable that contains the source code. For more information about here-strings, see about_Quoting_Rules[2].

Include a namespace declaration in your type definition. If you omit the namespace declaration, your type might have the same name as another type or the shortcut for another type, causing an unintentional overwrite. For example, if you define a type called "Exception", scripts that use "Exception" as the shortcut for System.Exception will fail.

Required?	true
Position?	1
Default Value	none
Accept Pipeline Input?	false
Accept Wildcard Characters?	false

-UsingNamespace <string[]>

Specifies other namespaces that are required for the class. This is much like the Using keyword in C#.

By default, Add-Type references the System namespace. When the MemberDefinition parameter is used, Add-Type also references the System.Runtime.InteropServices namespace by default. The namespaces that you add by using the UsingNamespace parameter are referenced in addition to the default namespaces.

Required?	false
Position?	named
Default Value	none

Accept Pipeline Input?	false
Accept Wildcard Characters?	false

<CommonParameters>

This command supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, WarningAction, and WarningVariable. For more information, see about_CommonParameters[3].

Inputs and Outputs

The input type is the type of the objects that you can pipe to the cmdlet. The return type is the type of the objects that the cmdlet returns.

Inputs	None You cannot pipe objects to Add-Type.
Outputs	None or System.RuntimeType When you use the PassThru parameter, Add-Type returns a System.RuntimeType object that represents the new type. Otherwise, this cmdlet does not generate any output.

Notes

The types that you add exist only in the current session. To use the types in all sessions, add them to your Windows PowerShell profile. For more information about the profile, see about_Profiles[4].

Type names (and namespaces) must be unique within a session. You cannot unload a type or change it. If you need to change the code for a type, you must change the name or start a new Windows PowerShell session. Otherwise, the command fails.

The CodeDomProvider class for some languages, such as IronPython and JSharp, does not generate output. As a result, types written in these languages cannot be used with Add-Type.

This cmdlet is based on the CodeDomProvider class. For more information about this class, see the Microsoft .NET Framework SDK.

Example 1

```
C:\PS>$source = @"
public class BasicTest
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }

    public int Multiply(int a, int b)
```

```
        {
            return (a * b);
        }
    }
    "@

    C:\PS> Add-Type -TypeDefinition $source

    C:\PS> [BasicTest]::Add(4, 3)

    C:\PS> $basicTestObject = New-Object BasicTest
    C:\PS> $basicTestObject.Multiply(5, 2)
```

Description

These commands add the BasicTest class to the session by specifying source code that is stored in a variable. The type has a static method called Add and a non-static method called Multiply.

The first command stores the source code for the class in the $source variable.

The second command uses the Add-Type cmdlet to add the class to the session. Because it is using inline source code, the command uses the TypeDefinition parameter to specify the code in the $source variable.

The remaining commands use the new class.

The third command calls the Add static method of the BasicTest class. It uses the double-colon characters (::) to specify a static member of the class.

The fourth command uses the New-Object cmdlet to instantiate an instance of the BasicTest class. It saves the new object in the $basicTestObject variable.

The fifth command uses the Multiply method of $basicTestObject.

Example 2

```
C:\PS>[BasicTest] | get-member

C:\PS> [BasicTest] | get-member -static

C:\PS> $basicTestObject | get-member

C:\PS> [BasicTest] | get-member


   TypeName: System.RuntimeType

Name                          MemberType Definition
----                          ---------- ----------
Clone                         Method     System.Object Clone()
Equals                        Method     System.Boolean Equals
FindInterfaces                Method     System.Type[] FindInt
...


C:\PS> [BasicTest] | get-member -static

   TypeName: BasicTest
```

```
Name            MemberType Definition
----            ---------- ----------
Add             Method     static System.Int32 Add(Int32 a, Int32 b)
Equals          Method     static System.Boolean Equals(Object objA,
ReferenceEquals Method     static System.Boolean ReferenceEquals(Obj


C:\PS> $basicTestObject | get-member

   TypeName: BasicTest

Name        MemberType Definition
----        ---------- ----------
Equals      Method     System.Boolean Equals(Object obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
Multiply    Method     System.Int32 Multiply(Int32 a, Int32 b)
ToString    Method     System.String ToString()
```

Description

These commands use the Get-Member cmdlet to examine the objects that the Add-Type and New-Object cmdlets created in the previous example.

The first command uses the Get-Member cmdlet to get the type and members of the BasicTest class that Add-Type added to the session. The Get-Member command reveals that it is a System.RuntimeType object, which is derived from the System.Object class.

The second command uses the Static parameter of Get-Member to get the static properties and methods of the BasicTest class. The output shows that the Add method is included.

The third command uses Get-Member to get the members of the object stored in the $BasicTestObject variable. This was the object instance that was created by using the New-Object cmdlet with the $BasicType class.

The output reveals that the value of the $basicTestObject variable is an instance of the BasicTest class and that it includes a member called Multiply.

Example 3

```
C:\PS>$accType = add-type -assemblyname accessib* -passthru
```

Description

This command adds the classes from the Accessibility assembly to the current session. The command uses the AssemblyName parameter to specify the name of the assembly. The wildcard character allows you to get the correct assembly even when you are not sure of the name or its spelling.

The command uses the PassThru parameter to generate objects that represent the classes that are added to the session, and it saves the objects in the $accType variable.

Example 4

```
C:\PS>add-type -path c:\ps-test\Hello.vb

[VBFromFile]::SayHello(", World")

# From Hello.vb
Public Class VBFromFile

Public Shared Function SayHello(sourceName As String) As String
Dim myValue As String = "Hello"

return myValue + sourceName
End Function
End Class

C:\PS> [VBFromFile]::SayHello(", World")
Hello, World
```

Description

This example uses the Add-Type cmdlet to add the VBFromFile class that is defined in the Hello.vb file to the current session. The text of the Hello.vb file is shown in the command output.

The first command uses the Add-Type cmdlet to add the type defined in the Hello.vb file to the current session. The command uses the path parameter to specify the source file.

The second command calls the SayHello function as a static method of the VBFromFile class.

Example 5

```
C:\PS>$signature = @"
[DllImport("user32.dll")]
public static extern bool ShowWindowAsync(IntPtr hWnd, int nCmdShow);
"@

$showWindowAsync = Add-Type -memberDefinition $signature -name "Win32ShowWindowAsync" -namespa
ce Win32Functions -passThru

# Minimize the Windows PowerShell console
$showWindowAsync::ShowWindowAsync((Get-Process -id $pid).MainWindowHandle, 2)

# Restore it
$showWindowAsync::ShowWindowAsync((Get-Process -id $pid).MainWindowHandle, 4)
```

Description

The commands in this example demonstrate how to call native Windows APIs in Windows PowerShell. Add-Type uses the Platform Invoke (P/Invoke) mechanism to call a function in User32.dll from Windows PowerShell.

The first command stores the C# signature of the ShowWindowAsync function in the $signature variable. (For more information, see "ShowWindowAsync Function" in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=143643.) To ensure that the resulting method will be visible in a Windows PowerShell session, the "public" keyword has been added to the standard signature.

The second command uses the Add-Type cmdlet to add the ShowWindowAsync function to the Windows PowerShell session as a static method of a class that Add-Type creates. The command uses the MemberDefinition parameter to specify the method definition saved in the $signature

variable.

The command uses the Name and Namespace parameters to specify a name and namespace for the class. It uses the PassThru parameter to generate an object that represents the types, and it saves the object in the $showWindowAsync variable.

The third and fourth commands use the new ShowWindowAsync static method. The method takes two parameters, the window handle, and an integer specifies how the window is to be shown.

The third command calls ShowWindowAsync. It uses the Get-Process cmdlet with the $pid automatic variable to get the process that is hosting the current Windows PowerShell session. Then it uses the MainWindowHandle property of the current process and a value of "2", which represents the SW_MINIMIZE value.

To restore the window, the fourth command use a value of "4" for the window position, which represents the SW_RESTORE value. (SW_MAXIMIZE is 3.)

Example 6

```
C:\PS>Add-Type -MemberDefinition $jsMethod -Name "PrintInfo" -Language JScript
```

Description

This command uses the Add-Type cmdlet to add a method from inline JScript code to the Windows PowerShell session. It uses the MemberDefinition parameter to submit source code stored in the $jsMethod variable. It uses the Name variable to specify a name for the class that Add-Type creates for the method and the Language parameter to specify the JScript language.

Example 7

```
C:\PS>Add-Type -Path FSharp.Compiler.CodeDom.dll


C:\PS> Add-Type -Path FSharp.Compiler.CodeDom.dll
C:\PS> $provider = New-Object Microsoft.FSharp.Compiler.CodeDom.FSharpCodeProvider

C:\PS> $fSharpCode = @"
let rec loop n =
if n <= 0 then () else begin
print_endline (string_of_int n);
loop (n-1)
end
"@

C:\PS> $fsharpType = Add-Type -TypeDefinition $fSharpCode -CodeDomProvider $provider -PassThru
 | where { $_.IsPublic }
C:\PS> $fsharpType::loop(4)
4
3
2
1
```

Description

This example shows how to use the Add-Type cmdlet to add an FSharp code compiler to your Windows PowerShell session. To run this example in Windows PowerShell, you must have the FSharp.Compiler.CodeDom.dll that is installed with the FSharp language.

The first command in the example uses the Add-Type cmdlet with the Path parameter to specify an assembly. Add-Type gets the types in the assembly.

The second command uses the New-Object cmdlet to create an instance of the FSharp code provider and saves the result in the $provider variable.

The third command saves the FSharp code that defines the Loop method in the $FSharpCode variable.

The fourth command uses the Add-Type cmdlet to save the public types defined in $fSharpCode in the $fSharpType variable. The TypeDefinition parameter specifies the source code that defines the types. The CodeDomProvider parameter specifies the source code compiler.

The PassThru parameter directs Add-Type to return a Runtime object that represents the types and a pipeline operator (|) sends the Runtime object to the Where-Object cmdlet, which returns only the public types. The Where-Object filter is used because the FSharp provider generates non-public types to support the resulting public type.

The fifth command calls the Loop method as a static method of the type stored in the $fSharpType variable.

See Also

Concepts

Add-Member[5]
New-Object[1]

Links Table

[1]*http://technet.microsoft.com/en-US/library/dd315334.aspx*

[2]*http://technet.microsoft.com/en-US/library/dd315325.aspx*

[3]*http://technet.microsoft.com/en-US/library/dd315352.aspx*

[4]*http://technet.microsoft.com/en-US/library/dd315342.aspx*

[5]*http://technet.microsoft.com/en-US/library/dd347695.aspx*

Community Content

© 2012 Microsoft. All rights reserved.