Full credit is given to the above companies including the OS that this PDF file was generated!

PowerShell Get-Help on command 'Add-Type'

PS C:\Users\wahid> Get-Help Add-Type

NAME

Add-Type

SYNOPSIS

Adds a Microsoft .NET class to a PowerShell session.

SYNTAX

Add-Type -AssemblyName <System.String[]> [-IgnoreWarnings] [-PassThru]
[<CommonParameters>]

Add-Type [-TypeDefinition] <System.String> [-CodeDomProvider
<System.CodeDom.Compiler.CodeDomProvider>] [-CompilerParameters
<System.CodeDom.Compiler.CompilerParameters>] [-IgnoreWarnings] [-Language
{CSharp | CSharpVersion2 | CSharpVersion3 | JScript | VisualBasic}]
[-OutputAssembly <System.String>] [-OutputType {ConsoleApplication | Library |
WindowsApplication}] [-PassThru] [-ReferencedAssemblies <System.String[]>]
[<CommonParameters>]

Add-Type [-Name] <System.String> [-MemberDefinition] <System.String[]>
[-CodeDomProvider <System.CodeDom.Compiler.CodeDomProvider>]

[-CompilerParameters <System.CodeDom.Compiler.CompilerParameters>]
[-IgnoreWarnings] [-Language {CSharp | CSharpVersion2 | CSharpVersion3 |
JScript | VisualBasic}] [-Namespace <System.String>] [-OutputAssembly
<System.String>] [-OutputType {ConsoleApplication | Library |
WindowsApplication}] [-PassThru] [-ReferencedAssemblies <System.String[]>]
[-UsingNamespace <System.String[]>] [<CommonParameters>]

Add-Type [-Path] <System.String[]> [-CompilerParameters
<System.CodeDom.Compiler.CompilerParameters>] [-IgnoreWarnings]
[-OutputAssembly <System.String>] [-OutputType {ConsoleApplication | Library |
WindowsApplication}] [-PassThru] [-ReferencedAssemblies <System.String[]>]
[<CommonParameters>]

Add-Type [-CompilerParameters <System.CodeDom.Compiler.CompilerParameters>]
[-IgnoreWarnings] -LiteralPath <System.String[]> [-OutputAssembly
<System.String>] [-OutputType {ConsoleApplication | Library |
WindowsApplication}] [-PassThru] [-ReferencedAssemblies <System.String[]>]
[<CommonParameters>]

DESCRIPTION

The `Add-Type` cmdlet lets you define a Microsoft .NET Framework class in your PowerShell session. You can then instantiate objects, by using the `New-Object` cmdlet, and use the objects just as you would use any .NET Framework object. If you add an `Add-Type` command to your PowerShell profile, the class is available in all 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. On Windows, you can use this feature to make Platform Invoke (P/Invoke) calls to unmanaged functions in 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, C# is the default, compiler options, assembly dependencies, the class namespace, the names of the type, and the resulting assembly.

PARAMETERS

-AssemblyName <System.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're 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.

Using the Path or LiteralPath parameters guarantees that you are loading the assembly that you intended to load. When you use the AssemblyName parameter, PowerShell asks .NET to resolve the assembly name using the standard .NET assembly resolution process. Since .NET searches the application folder first, `Add-Type` might load an assembly from `$PSHOME` instead of the version in the current folder. For more information, see Assembly location (/dotnet/standard/assembly/location).

If .NET fails to resolve the name, PowerShell then looks in the current location to find the assembly. When you use wildcards in the AssemblyName parameter, the .NET assembly resolution process fails causing PowerShell to look in the current location.

-CodeDomProvider <System.CodeDom.Compiler.CodeDomProvider>

Specifies a code generator or compiler. `Add-Type` uses the specified

compiler to compile the source code. The default is the C# compiler. Use this parameter if you're using a language that can't be specified by using the Language parameter. The CodeDomProvider that you specify must be able to generate assemblies from source code.

-CompilerParameters <System.CodeDom.Compiler.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 .

You can't use the CompilerParameters and ReferencedAssemblies parameters in the same command.

-IgnoreWarnings <System.Management.Automation.SwitchParameter>
Ignores compiler warnings. Use this parameter to prevent `Add-Type` from handling compiler warnings as errors.

-Language <Microsoft.PowerShell.Commands.Language>
Specifies the language that is used in the source code. The `Add-Type` cmdlet uses the value of this parameter to select the appropriate CodeDomProvider . CSharp is the default value. The acceptable values for this parameter are as follows:

- `CSharp`

- `CSharpVersion2`

- `CSharpVersion3`

- `JScript`

- `VisualBasic`

-LiteralPath <System.String[]>

Specifies the path to source code files or assembly DLL files that contain the types. Unlike Path , the value of the LiteralPath parameter is used exactly as it's typed. No characters are interpreted as wildcards. If the path includes escape characters, enclose it in single quotation marks. Single quotation marks tell PowerShell not to interpret any characters as escape sequences.

Using the Path or LiteralPath parameters guarantees that you are loading the assembly that you intended to load.

-MemberDefinition <System.String[]>

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

On Windows, you can use this feature to make Platform Invoke (P/Invoke) calls to unmanaged functions in PowerShell.

-Name <System.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 can't unload a type or change it. To change the code for a type, you must change the name or start a new PowerShell session. Otherwise, the command fails.

-Namespace <System.String>

Specifies a namespace for the type.

If this parameter isn't 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.

-OutputAssembly <System.String>

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

-OutputType <Microsoft.PowerShell.Commands.OutputAssemblyType>

Specifies the output type of the output assembly. By default, no output type is specified. This parameter is valid only when an output assembly is specified in the command. For more information about the values, see OutputAssemblyType Enumeration (/dotnet/api/microsoft.powershell.commands.outputassemblytype).

The acceptable values for this parameter are as follows:

- `ConsoleApplication`

- `Library`

- `WindowsApplication`

-PassThru <System.Management.Automation.SwitchParameter>

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

-Path <System.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 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.

-ReferencedAssemblies <System.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.

You can't use the CompilerParameters and ReferencedAssemblies parameters in the same command.

-TypeDefinition <System.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 (../Microsoft.PowerShell.Core/about/about_Quoting_Rules.md).

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.

-UsingNamespace <System.String[]>

Specifies other namespaces that are required for the class. This is much

like the C# keyword, `Using`.

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.

<CommonParameters>
This cmdlet supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer, PipelineVariable, and OutVariable. For more information, see about_CommonParameters (https:/go.microsoft.com/fwlink/?LinkID=113216).

---------- Example 1: Add a .NET type to a session ----------

```
$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);
    }
}
"@

Add-Type -TypeDefinition $Source
[BasicTest]::Add(4, 3)
$BasicTestObject = New-Object BasicTest
```

$BasicTestObject.Multiply(5, 2)

The `$Source` variable stores the source code for the class. The type has a static method called `Add` and a non-static method called `Multiply`.

The `Add-Type` cmdlet adds the class to the session. Because it's using inline source code, the command uses the TypeDefinition parameter to specify the code in the `$Source` variable.

The `Add` static method of the BasicTest class uses the double-colon characters (`::`) to specify a static member of the class. The integers are added and the sum is displayed.

The `New-Object` cmdlet instantiates an instance of the BasicTest class. It saves the new object in the `$BasicTestObject` variable.

`$BasicTestObject` uses the `Multiply` method. The integers are multiplied and the product is displayed.

---------- Example 2: Examine an added type ----------

[BasicTest] | Get-Member

TypeName: System.RuntimeType

Name MemberType Definition

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

AsType Method type AsType()

Clone Method System.Object Clone(), System.Object ICloneable.Clone()

Equals Method bool Equals(System.Object obj), bool Equals(type o)

FindInterfaces Method type[] FindInterfaces(System.Reflection.TypeFilter filter...

...

```
[BasicTest] | Get-Member -Static

TypeName: BasicTest

Name           MemberType Definition
----           ---------- ----------
Add            Method     static int Add(int a, int b)
Equals         Method     static bool Equals(System.Object objA,
System.Object objB)
new            Method     BasicTest new()
ReferenceEquals Method     static bool ReferenceEquals(System.Object objA,
System.Object objB)

$BasicTestObject | Get-Member

TypeName: BasicTest

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
Multiply    Method     int Multiply(int a, int b)
ToString    Method     string ToString()
```

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

The `Get-Member` Static parameter gets the static properties and methods of the BasicTest class. The output shows that the `Add` method is included.

The `Get-Member` cmdlet gets the members of the object stored in the `$BasicTestObject` variable. `$BasicTestObject` was created by using the `New-Object` cmdlet with the BasicTest 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: Add types from an assembly ----------

```
$AccType = Add-Type -AssemblyName "accessib*" -PassThru
```

The `$AccType` variable stores an object created with the `Add-Type` cmdlet. `Add-Type` uses the AssemblyName parameter to specify the name of the assembly. The asterisk (`*`) wildcard character allows you to get the correct assembly even when you aren't sure of the name or its spelling. The PassThru parameter generates objects that represent the classes that are added to the session.

---------- Example 4: Call native Windows APIs ----------

```
$Signature = @"
[DllImport("user32.dll")]public static extern bool ShowWindowAsync(IntPtr
hWnd, int nCmdShow);
"@

$addTypeSplat = @{
   MemberDefinition = $Signature
   Name = "Win32ShowWindowAsync"
   Namespace = 'Win32Functions'
   PassThru = $true
}
$ShowWindowAsync = Add-Type @addTypeSplat

# Minimize the PowerShell console
```

```
$ShowWindowAsync::ShowWindowAsync((Get-Process -Id $pid).MainWindowHandle, 2)

# Restore the PowerShell console

$ShowWindowAsync::ShowWindowAsync((Get-Process -Id $Pid).MainWindowHandle, 4)
```

The `$Signature` variable stores the C# signature of the `ShowWindowAsync` function. To ensure that the resulting method is visible in a PowerShell session, the `public` keyword was added to the standard signature. For more information, see ShowWindowAsync (/windows/win32/api/winuser/nf-winuser-showwindowasync)function.

The `$ShowWindowAsync` variable stores the object created by the `Add-Type` PassThru parameter. The `Add-Type` cmdlet adds the `ShowWindowAsync` function to the PowerShell session as a static method. The command uses the MemberDefinition parameter to specify the method definition saved in the `$Signature` variable. The command uses the Name and Namespace parameters to specify a name and namespace for the class. The PassThru parameter generates an object that represents the types.

The new `ShowWindowAsync` static method is used in the commands to minimize and restore the PowerShell console. The method takes two parameters: the window handle, and an integer that specifies how the window is displayed.

To minimize the PowerShell console, `ShowWindowAsync` uses the `Get-Process` cmdlet with the `$PID` automatic variable to get the process that is hosting the current 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, `ShowWindowAsync` uses a value of `4` for the window position, which represents the `SW_RESTORE` value.

To maximize the window, use the value of `3` that represents `SW_MAXIMIZE`.

-------- Example 5: Add a type from a Visual Basic file --------

```
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
```

Hello, World

`Add-Type` uses the Path parameter to specify the source file, `Hello.vb`, and adds the type defined in the file. The `SayHello` function is called as a static method of the VBFromFile class.

---------- Example 6: Add a class with JScript.NET ----------

```
Add-Type @'
 class FRectangle {
   var Length : double;
   var Height : double;
   function Perimeter() : double {
       return (Length + Height) * 2; }
   function Area() : double {
       return Length * Height;  } }
'@ -Language JScript

$rect = [FRectangle]::new()
```

```
$rect

Length Height
------ ------
     0      0



---------- Example 7: Add an F# compiler ----------

Add-Type -Path "FSharp.Compiler.CodeDom.dll"
$Provider = New-Object Microsoft.FSharp.Compiler.CodeDom.FSharpCodeProvider
$FSharpCode = @"
let rec loop n =if n <= 0 then () else beginprint_endline (string_of_int
n);loop (n-1)end
"@
$FSharpType = Add-Type -TypeDefinition $FSharpCode -CodeDomProvider $Provider
-PassThru |
   Where-Object { $_.IsPublic }
$FSharpType::loop(4)

4
3
2
1
```

`Add-Type` uses the Path parameter to specify an assembly and gets the types in the assembly. `New-Object` creates an instance of the F# code provider and saves the result in the `$Provider` variable. The `$FSharpCode` variable saves the F# code that defines the Loop method.

The `$FSharpType` variable stores the results of the `Add-Type` cmdlet that saves the public types defined in `$FSharpCode`. 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. The objects are sent down the pipeline to the `Where-Object` cmdlet, which returns only the public types. The `Where-Object` cmdlet is used because the F# provider generates non-public types to support the resulting public type.

The Loop method is called as a static method of the type stored in the `$FSharpType` variable.

REMARKS

To see the examples, type: "get-help Add-Type -examples".

For more information, type: "get-help Add-Type -detailed".

For technical information, type: "get-help Add-Type -full".

For online help, type: "get-help Add-Type -online"