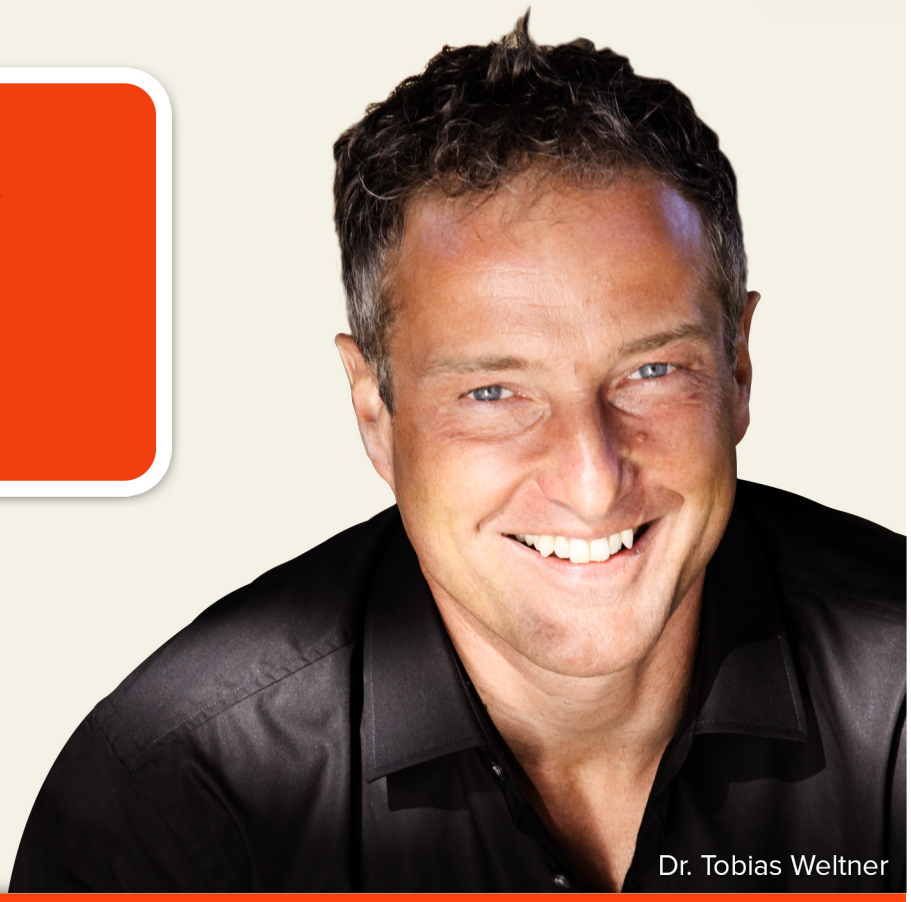
PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 4 | September 2013

This Month's Topic:

Objects & Types



Dr. Tobias Weltner

Table of Contents

1. Discovering Objects and Types

By default, PowerShell automatically determines the type when you assign a value to a variable:

```
PS> $a = 1

PS> $a.GetType().FullName
System.Int32

PS> $a = 1.87

PS> $a.GetType().FullName
System.Double

PS> $a = 623876378232

PS> $a.GetType().FullName
System.Int64

PS> $a = 'Hello'

PS> $a.GetType().FullName
System.String

PS> $a = Get-Date

PS> $a.GetType().FullName
System.DateTime
```

2. Assigning Better Data Types

If you want to assign a better data type, you may want to know the numeric ranges a given numeric data type can store. Here's how you can find this range:

```
PS> [Int32]::MaxValue
2147483647

PS> [Int64]::MaxValue
9223372036854775807

PS> [Byte]::MaxValue
255
```

This will also explain why there are unsigned data types. They do not allow negative values, and because of this, they allow for a larger positive range:

```
PS> [Int32]::MaxValue
2147483647

PS> [UInt32]::MaxValue
4294967295

PS> [Int32]::MinValue
-2147483648

PS> [UInt32]::MinValue
0
```

As you can see, unsigned types simply "shift" the negative range into the positive range. So if you do not need negative values, then unsigned data types provide a twice-as-large positive range.

To assign your own data type to a variable, simply prepend the data type:

```
PS> [Byte]$a = 100

PS> $a.GetType().FullName
System.Byte
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

3. Useful Data Conversions

PowerShell automatically picks a type for the data you use. However, PowerShell may not always pick the best--most specific--data type, simply because PowerShell cannot always know what the data "means". For example, you may want a number to be interpreted as ASCII code for a character. Simply convert the number to the Char type. This type represents one character:

```
PS> [Char]65
A
```

To do the opposite and convert a character to its ASCII value, use this:

```
PS> [Byte][Char]'A'
65
```

Here, the letter "A" is first converted into a Char type, then into a Byte value.

To process more than one character at a time, use arrays instead:

```
PS> [Byte[]][Char[]]'Hello'
72
101
108
108
111
```

And this gets you a range of letters:

```
[Char[]](65..90)
```

Here are some common examples for more specific data types and why it may be useful to manually convert data to such a type:

```
PS> [System.Net.Mail.MailAddress]'Some User<some.person@somewhere.com>'

DisplayName   User          Host           Address
-----------   ----          ----           -------
Some User     some.person   somewhere.com  some.person@somewhere.com
```

```
PS> [System.Net.Mail.MailAddress]'some.person@somewhere.com'

DisplayName   User          Host          Address
-----------   ----          ----          -------
              some.person   somewhere.com  some.person@somewhere.com


PS> ([System.Net.Mail.MailAddress]'some.person@somewhere.com').Host
somewhere.com

PS> [System.Net.IPAddress]'192.168.2.1'


Address            : 16951488
AddressFamily      : InterNetwork
ScopeId            :
IsIPv6Multicast    : False
IsIPv6LinkLocal    : False
IsIPv6SiteLocal    : False
IsIPv6Teredo       : False
IsIPv4MappedToIPv6 : False
IPAddressToString  : 192.168.2.1


PS> [System.Version]'10.1.3.22'

Major  Minor  Build  Revision
-----  -----  -----  --------
10     1      3      22



PS> ([System.Version]'10.1.3.22').Major
10

PS> ([System.Version]'10.1.3.22').Build
3


PS> [Char[]]'Hello'
H
e
l
l
o

PS> [Byte[]][Char[]]'Hello'
72
101
108
108
111
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

4. Chaining Type Conversions

In PowerShell, you can do multiple sequential type conversions. For example, you should first convert the string into a character array and then into the byte array to split a string into a byte array:

```
[Byte[]][Char[]]"Hello World!"
```

Now, why would that be useful? For example, you could write text as binary into your Registry:

```
$ByteArray = [Byte[]][Char[]]'MyProduct'

$null = New-Item -Path HKCU:\Software\Test -Force
Set-ItemProperty HKCU:\Software\Test ProductName -Type Binary -Value $ByteArray
```

5. Sorting Specific Types

Sort-Object typically accepts the type of the data you want to sort. This is why these two lines produce different results:

```
PS> 1,10,3,2 | Sort-Object
1
2
3
10

PS> '1','10','3','2' | Sort-Object
1
10
2
3
```

Using type conversion, you can fix this and tell Sort-Object to use a different type for sorting:

```
PS> '1','10','3','2' | Sort-Object -Property { [Double]$_ }
1
2
3
10
```

You can even use Sort-Object to produce random sorting like this:

```
PS> 1..10 | Sort-Object -Property { Get-Random }
7
1
5
3
8
10
9
2
4
6
```

And even hard-to-sort items like IP addresses can now be sorted just fine:

```
PS> '1.2.3.4','10.10.10.1','2.3.4.5', '110.12.1.1' | Sort-Object
1.2.3.4
10.10.10.1
110.12.1.1
2.3.4.5

PS> '1.2.3.4','10.10.10.1','2.3.4.5', '110.12.1.1' | Sort-Object -Property { [System.Version]$_ }
1.2.3.4
2.3.4.5
10.10.10.1
110.12.1.1
```

Here, the strings are converted to version numbers. Version numbers are not IP addresses, but they also consist of four numeric items, so Sort-Object will sort correctly.

6. Using Converter Class

Types can also have built-in methods and properties. They are called "static" because they are immediately available and do not require using New-Object to derive objects from a type.

You can access those with two colons. The type System.Convert, for example, can convert data into other types:

```
PS> $value = [System.Convert]::ToByte(123)

PS> $value.GetType().FullName
System.Byte

PS> $value = [System.Convert]::ToUInt64(123)

PS> $value.GetType().FullName
System.UInt64

PS> # binary string representation
PS> [System.Convert]::ToString(123,2)
1111011

PS> # hex string representation
PS> [System.Convert]::ToString(123,16)
7b

PS> # octal string representation
PS> [System.Convert]::ToString(123,8)
173
```

The opposite works, too. Use this approach to convert a binary to a decimal:

```
PS> $binary = '1110111000010001'
PS> [System.Convert]::ToInt64($binary, 2)
60945
```

7. Converting to Signed Number

If you convert a hex to a decimal number, the result may not be what you want:

```
PS> 0xFFFF
65535
```

PowerShell converts it to an unsigned number (unless its value is too large for an unsigned integer). If you need the signed number, you would have to use the BitConverter type and first make the hex number a byte array, then convert this back to a signed integer like this:

```
PS> [BitConverter]::ToInt16([BitConverter]::GetBytes(0xFFFF), 0)
-1
```

8. Converting to Signed Number Using Casting

In a previous tip, you learned how to use the BitConverter type to convert hexadecimals to signed integers. Here is another way that uses type conversion:.

```
PS> 0xffff
65535

PS> 0xfffe
65534

PS> [int16]("0x{0:x4}" -f ([UInt32]0xffff))
-1

PS> [int16]("0x{0:x4}" -f ([UInt32]0xfffe))
-2
```

9. Converting Letters to Numbers and Bit Masks

Sometimes, you may need to turn characters such as drive letters into numbers (or even bit masks) that you then can use to hide certain drives in Windows Explorer.

With a little bit of math, this is easily doable. Here is a sample:

Let's start with an unsorted list of drive letters, and then turn it into an ordered bit mask:

```
$DriveList = 'a', 'b:', 'd', 'Z', 'x:'

$DriveList |
    ForEach-Object { $_.ToUpper()[0] } |
    Sort-Object
```

This gets you a sorted and normalized list of drive letters. Next, add one more pipeline element to turn the letters into bit numbers:

```
$DriveList = 'a', 'b:', 'd', 'Z', 'x:'

$DriveList |
    ForEach-Object { $_.ToUpper()[0] } |
    Sort-Object |
    ForEach-Object { ([Byte]$_) -65 }
```

To turn this into a bit mask, use the Pow() function:

```
$DriveList = 'a', 'b:', 'd', 'Z', 'x:'

$DriveList |
    ForEach-Object { $_.ToUpper()[0] } |
    Sort-Object |
    ForEach-Object { [Math]::Pow(2,(([Byte]$_) -65)) }
```

10. Finding Static Methods

The methods and properties provided by types are called "static" (because you do not need to generate an object first). PowerShell ISE shows static methods in its IntelliSense menu when you add "::" to a type:

```
[DateTime]::
```

To get the same information as a list, pass the type to Get-Member, but do not forget to specify the -Static switch. Without it, you won't see the static members, but instead just the instance members of the type (which describe the type itself):

```
[DateTime] | Get-Member -Static
[DateTime]::IsLeapYear(2010)
```

11. Using Static Properties

Once you found an interesting static property in a type, here is how you find out how to call it.

Let's assume you found the type System.Environment. This line gives you all the static properties:

```
PS> [System.Environment] | Get-Member -Static -MemberType *property


   TypeName: System.Environment

Name                      MemberType Definition
----                      ---------- ----------
CommandLine               Property   static string CommandLine {get;}
CurrentDirectory          Property   static string CurrentDirectory {get;set;}
CurrentManagedThreadId    Property   static int CurrentManagedThreadId {get;}
ExitCode                  Property   static int ExitCode {get;set;}
HasShutdownStarted        Property   static bool HasShutdownStarted {get;}
Is64BitOperatingSystem    Property   static bool Is64BitOperatingSystem {get;}
Is64BitProcess            Property   static bool Is64BitProcess {get;}
MachineName               Property   static string MachineName {get;}
NewLine                   Property   static string NewLine {get;}
OSVersion                 Property   static System.OperatingSystem OSVersion {g...
ProcessorCount            Property   static int ProcessorCount {get;}
StackTrace                Property   static string StackTrace {get;}
SystemDirectory           Property   static string SystemDirectory {get;}
SystemPageSize            Property   static int SystemPageSize {get;}
TickCount                 Property   static int TickCount {get;}
UserDomainName            Property   static string UserDomainName {get;}
UserInteractive           Property   static bool UserInteractive {get;}
UserName                  Property   static string UserName {get;}
Version                   Property   static version Version {get;}
WorkingSet                Property   static long WorkingSet {get;}
```

A property is just a piece of information that you can retrieve like a variable. This gets you the current system directory:

```
PS> [System.Environment]::SystemDirectory
C:\Windows\system32
```

If a property is labeled “get;set;”, then you can also change it. To change the current system directory, try this:

```
PS> [System.Environment]::CurrentDirectory = ‘c:\’

PS> [System.Environment]::CurrentDirectory
c:\
```

The current system directory is not equal to PowerShell’s current path. The current system directory is the default directory used by .NET methods. PowerShell still uses its own current path for cmdlets:

```
PS> Get-Location

Path
----
C:\Users\Tobias
```

There are plenty of useful pieces of information. This provides information about your operating system:

```
PS> [System.Environment]::OSVersion

Platform ServicePack         Version             VersionString
-------- -----------         -------             -------------
Win32NT  Service Pack 1      6.1.7601.65536      Microsoft Window...


PS> [System.Environment]::OSVersion.ServicePack
Service Pack 1

PS> [System.Environment]::OSVersion.Version

Major  Minor  Build  Revision
-----  -----  -----  --------
6      1      7601   65536


PS> [System.Environment]::Is64BitOperatingSystem
True

PS> [System.Environment]::Is64BitProcess
True

PS> [System.Environment]::MachineName
TOBIASAIR1

PS> [System.Environment]::TickCount
609016093
```

The TickCount property, for example, returns the number of "ticks" since your machine was started, which you could use as a high-resolution timer to measure how long it takes to execute parts of your code:

```
$start = [System.Environment]::TickCount
Start-Sleep -Seconds 2
$end = [System.Environment]::TickCount

$duration = $end-$start

"Milliseconds: $duration ms"
```

12. Using Static Methods

Let's assume you found the type System.Environment. This line gives you all the static methods:

```
PS> [System.Environment] | Get-Member -Static -MemberType *method


   TypeName: System.Environment

Name                      MemberType Definition
----                      ---------- ----------
Equals                    Method     static bool Equals(System.Object objA,...
Exit                      Method     static void Exit(int exitCode)
ExpandEnvironmentVariables Method     static string ExpandEnvironmentVariabl...
FailFast                  Method     static void FailFast(string message), ...
GetCommandLineArgs        Method     static string[] GetCommandLineArgs()
GetEnvironmentVariable    Method     static string GetEnvironmentVariable(s...
GetEnvironmentVariables   Method     static System.Collections.IDictionary ...
GetFolderPath             Method     static string GetFolderPath(System.Env...
GetLogicalDrives          Method     static string[] GetLogicalDrives()
ReferenceEquals           Method     static bool ReferenceEquals(System.Obj...
SetEnvironmentVariable    Method     static void SetEnvironmentVariable(str...
```

Methods are commands, and they always require brackets. If a method expects arguments, then these arguments are specified inside the brackets as a comma-separated list. To find out just which arguments a method needs, look at the definition column.

Let's assume you want to permanently set an environment variable. You have spotted SetEnvironmentVariable() method. This gets you the method signature (the list of required arguments):

```
PS> [System.Environment] | Get-Member -Static -Name SetEnvironmentVariable | Select-Object
-ExpandProperty Definition
static void SetEnvironmentVariable(string variable, string value), static void
SetEnvironmentVariable(string variable, string value, System.EnvironmentVariableTarget
 target)
```

A much easier approach uses a trick: just omit the method brackets:

```
PS> [System.Environment]::SetEnvironmentVariable
```

```
OverloadDefinitions
-------------------
static void SetEnvironmentVariable(string variable, string value)
static void SetEnvironmentVariable(string variable, string value,
System.EnvironmentVariableTarget target)
```

As you can see, this method requires two or three arguments. Sometimes, the name of the arguments lets you guess what they mean. To set an environment variable named “test” with the value “Hello” in the user context, try this:

```
PS> [System.Environment]::SetEnvironmentVariable('test','hello','user')
```

If cannot guess, look at the data types for the arguments. The first two are strings, the third is of type System.EnvironmentVariableTarget. To find out which values are permitted for this type, try this:

```
PS> [System.Enum]::GetNames([System.EnvironmentVariableTarget])
Process
User
Machine
```

Since most .NET types are very well documented, you can also navigate to your favorite Internet search engine, and search for “System. Environment SetEnvironmentVariable”. This gets you quickly the full documentation.

When you apply this knowledge to GetFolderPath, this is how you find the signature:

```
PS> [System.Environment]::GetFolderPath

OverloadDefinitions
-------------------
static string GetFolderPath(System.Environment+SpecialFolder folder)
static string GetFolderPath(System.Environment+SpecialFolder folder,
System.Environment+SpecialFolderOption option)
```

To find out what the argument wants, look at its type:

```
PS> [System.Enum]::GetNames([System.Environment+SpecialFolder])
Desktop
Programs
MyDocuments
Personal
Favorites
Startup
Recent
SendTo
StartMenu
MyMusic
MyVideos
DesktopDirectory
MyComputer
```

```
NetworkShortcuts
Fonts
Templates
CommonStartMenu
CommonPrograms
CommonStartup
CommonDesktopDirectory
ApplicationData
PrinterShortcuts
LocalApplicationData
InternetCache
Cookies
History
CommonApplicationData
Windows
System
ProgramFiles
MyPictures
UserProfile
SystemX86
ProgramFilesX86
CommonProgramFiles
CommonProgramFilesX86
CommonTemplates
CommonDocuments
CommonAdminTools
AdminTools
CommonMusic
CommonPictures
CommonVideos
Resources
LocalizedResources
CommonOemLinks
CDBurning
```

Now you know how to find out system folder paths. If you wanted to find the path to the common music folder on a particular machine, this is what you do:

```
PS> [System.Environment]::GetFolderPath('CommonMusic')
C:\Users\Public\Music
```

13. Resolving Host Name

The type System.Net.DNS provides methods to query DNS services:

```
[System.Net.Dns]::GetHostByName('microsoft.com')
[System.Net.Dns]::GetHostByAddress('10.10.12.100')
[System.Net.Dns]::GetHostByName()
```

14. Stripping Decimals without Rounding

When you divide numbers and just want the decimals before the decimal point, you could cast the result to integer. However, this would also round the result:

```
PS> 18 / 5
3,6

PS> [Int](18/5)
4
```

The Math type has all the advanced math functions needed to strip decimals, explicitly round up or down, along with many more:

```
PS> [Math]::Truncate(18/5)
3

PS> [Math]::Ceiling(3.1)
4

PS> [Math]::Floor(3.9)
3
```

15. Generate a New GUID

GUIDs are “Globally Unique Identifiers” which are so random that you can safely assume they are unique worldwide. GUIDs are used whenever you want to make sure you get a truly unique ID. Use them to identify components or generate unique file names. Here is how you generate new GUIDs:

```
PS> [System.Guid]::NewGuid().ToString()
5a14248f-b696-4948-9a89-52800092f77e

PS> [System.Guid]::NewGuid().ToString()
b5f21a9e-669c-4367-b334-5a02ee032ae1
```

16. Get .NET Runtime Directory

To find out where your .NET runtime folder is, try this line:

```
PS> [System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory()
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
```

17. Extracting Icons with PowerShell

To extract an icon from a file, use .NET Framework methods. The type System.Drawing assembly has a method called "ExtractAssociatedIcon" to get the icon that is associated with a file. This type is loaded automatically by the ISE editor, but the PowerShell console does not need it, so it is not available here. Which is why you should always make sure the type is loaded. Have Add-Type load the assembly first.

Here is a sample that extracts the powershell.exe default icon and saves it as ICO file:

```
Add-Type -AssemblyName System.Drawing

$FilePath = "$pshome\powershell.exe"
$IconPath = "$env:temp\powershell.ico"
[System.Drawing.Icon]::ExtractAssociatedIcon($FilePath).ToBitmap().Save($IconPath)

explorer "/select,$IconPath"
```

18. Creating Your Own Type

Did you know that you can compile any .NET source code on the fly and use this to create your own types?

Here is an example illustrating how to create a new type from C# code that has both static and dynamic methods:

```
$source = @'
public class Calculator
{
      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
[Calculator]::Add(5,10)
$myCalculator = New-Object Calculator
$myCalculator.Multiply(5,10)
```

A static method (like Add() in this example) is accessible through the type. A dynamic method (like Multiply() in this example) requires an object that is derived from the type using New-Object.

So whether a member is accessible through a type or rather through an object derived from that type is solely a decision the developer makes. It is typically based on the question: is my method generic, or is it specific to some instance or situation? Since there is always just one type, but there can be any number of objects, static members typically do generic things whereas dynamic members act on the data that is contained in an object.

19. Create Custom Enumerations

PowerShell cannot create new types, but the cmdlet Add-Type can - it can use C# code to define a new type and make it available within PowerShell.

So if you want to create your own list of allowed values, you can do so:

```
$source = @'
public enum DayTime
{
    Morning=1,
    Noon = 2,
    Afternoon = 3,
    Evening = 4,
    Night = 5
}
'@
Add-Type -TypeDefinition $source
```

This creates a new type called “DayTime” that defines a list of allowed values.

When you apply this type to a variable, it now can only contain the values you specified:

```
PS> [DayTime]$a = ‘Noon’

PS> [DayTime]$a = ‘Evening’

PS> [DayTime]$a = ‘Midnight’
Cannot convert value “Midnight” to type “DayTime”. Error: “Unable to match the
identifier name Midnight to a valid enumerator name.  Specify one of the
following enumerator names and try again: Morning, Noon, Afternoon, Evening,
Night”
```

If you try and assign a value that cannot be converted into any one of the allowed values, the error message lists the allowed values at the end of the error message text.

When you assign the type to a function parameter, PowerShell ISE will now even show IntelliSense for the parameter, offering the allowed names in its IntelliSense menu:

```
function test
{
  param
  (
    [DayTime]
    $TheDayTime
  )
}
```

20. Finding Existing Enumeration Data Types

You do not necessarily have to create your own enumerations just to tie a variable to a list of legal values. The .NET Framework already comes with zillions of types that you can borrow. You just need to know the name of the appropriate type.

Here is some code that’ll find and list all enumeration data types available in your PowerShell session:

```
[AppDomain]::CurrentDomain.GetAssemblies() |
  ForEach-Object { trap{continue} $_.GetExportedTypes() } |
  Where-Object { $_.isEnum } |
  Sort-Object FullName |
  ForEach-Object {
    $values = [System.Enum]::GetNames($_) -join ‘,’
    $rv = $_ | Select-Object -Property FullName, Values
    $rv.Values = $values
    $rv
  }
```

You get back two columns: on the left side the name of the Enum data type and on the right side the list of values defined by the Enum. Here’s a short part of this:

```
System.Xml.XmlOutputMethod              Xml,Html,Text,AutoDetect
System.Xml.XmlSpace                     None,Default,Preserve
System.Xml.XmlTokenizedType             CDATA,ID,IDREF,IDREFS,ENTITY,ENTITIE...
System.Xml.XPath.XmlCaseOrder           None,UpperFirst,LowerFirst
System.Xml.XPath.XmlDataType            Text,Number
System.Xml.XPath.XmlSortOrder           Ascending,Descending
System.Xml.XPath.XPathNamespaceScope    All,ExcludeXml,Local
```

If you want to limit a variable or a parameter to the numbers listed by a particular Enum, just use this type with the variable or parameter.

```
function Test-Parameter
{
    param
    (
        [System.Xml.XmlOutputMethod]
        $Format,

        [System.Xml.XPath.XmlDataType]
        $Type,

        [System.Xml.XPath.XmlSortOrder]
        $SortOrder
    )

    "some things I could do now..."
}
```

This function "borrows" three types and decorates its parameters with them. In PowerShell 3.0, ISE will now show IntelliSense menus when you use this function, and the console will support TAB expansion.

21. Get List of Type Accelerators

Ever wondered what the difference between [Int], [Int32], and [System.Int32] is? They all are data types, and the first two are type accelerators, so they are really all the same.

```
PS> [Int].FullName
System.Int32
```

To list all the type accelerators PowerShell provides, use this undocumented (and unsupported) call:

```
[PSObject].Assembly.GetType("System.Management.Automation.TypeAccelerators")::Get
```

22. Working with Objects

Any result you get back from PowerShell really is an object. It has properties and methods, just like types do. To access them, use "." (dot-notation).

This line creates a text file and then accesses it using Get-Item:

```
$Path = "$env:temp\SomeFile.txt"
"Hello" > $Path

$file = Get-Item -Path $Path
$file
```

When you run this, $file returns the text representation of the file. It is a real object, though, and here is how you can find out all properties and methods of it:

```
PS> $file | Get-Member


   TypeName: System.IO.FileInfo

Name                      MemberType     Definition
----                      ----------     ----------
Mode                      CodeProperty   System.String Mode{get=Mode;}
AppendText                Method         System.IO.StreamWriter AppendText()
CopyTo                    Method         System.IO.FileInfo CopyTo(string de...
Create                    Method         System.IO.FileStream Create()
CreateObjRef              Method         System.Runtime.Remoting.ObjRef Crea...
CreateText                Method         System.IO.StreamWriter CreateText()
Decrypt                   Method         void Decrypt()
Delete                    Method         void Delete()
Encrypt                   Method         void Encrypt()
Equals                    Method         bool Equals(System.Object obj)
GetAccessControl          Method         System.Security.AccessControl.FileS...
GetHashCode               Method         int GetHashCode()
GetLifetimeService        Method         System.Object GetLifetimeService()
GetObjectData             Method         void GetObjectData(System.Runtime.S...
GetType                   Method         type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetimeSer...
MoveTo                    Method         void MoveTo(string destFileName)
Open                      Method         System.IO.FileStream Open(System.IO...
OpenRead                  Method         System.IO.FileStream OpenRead()
OpenText                  Method         System.IO.StreamReader OpenText()
OpenWrite                 Method         System.IO.FileStream OpenWrite()
Refresh                   Method         void Refresh()
Replace                   Method         System.IO.FileInfo Replace(string d...
SetAccessControl          Method         void SetAccessControl(System.Securi...
ToString                  Method         string ToString()
PSChildName               NoteProperty   System.String PSChildName=SomeFile.txt
PSDrive                   NoteProperty   System.Management.Automation.PSDriv...
PSIsContainer             NoteProperty   System.Boolean PSIsContainer=False
PSParentPath              NoteProperty   System.String PSParentPath=Microsof...
PSPath                    NoteProperty   System.String PSPath=Microsoft.Powe...
PSProvider                NoteProperty   System.Management.Automation.Provid...
Attributes                Property       System.IO.FileAttributes Attributes...
CreationTime              Property       datetime CreationTime {get;set;}
CreationTimeUtc           Property       datetime CreationTimeUtc {get;set;}
Directory                 Property       System.IO.DirectoryInfo Directory {...
DirectoryName             Property       string DirectoryName {get;}
Exists                    Property       bool Exists {get;}
Extension                 Property       string Extension {get;}
FullName                  Property       string FullName {get;}
IsReadOnly                Property       bool IsReadOnly {get;set;}
LastAccessTime            Property       datetime LastAccessTime {get;set;}
LastAccessTimeUtc         Property       datetime LastAccessTimeUtc {get;set;}
LastWriteTime             Property       datetime LastWriteTime {get;set;}
LastWriteTimeUtc          Property       datetime LastWriteTimeUtc {get;set;}
Length                    Property       long Length {get;}
Name                      Property       string Name {get;}
BaseName                  ScriptProperty System.Object BaseName {get=if ($th...
VersionInfo               ScriptProperty System.Object VersionInfo {get=[Sys...
```

So this would get you the file creation time:

```
PS> $file.CreationTime

Sunday, July 28, 2013 1:38:00 PM
```

And this would encrypt the file with the EFS. In Explorer, the file now appears “green” and can only be opened by the person that encrypted it:

```
$file.Encrypt()
```

Decrypt() will decrypt it again. Note how methods (commands) always require brackets:

```
$file.Decrypt()
```

Some properties can also be changed. They are marked with “get;set;”. So if you wanted to change the creation time, here is how:

```
$file.CreationTime = ‘1622-12-01 06:22:10’
explorer “/select,””$Path”””
```

The Explorer opens and selects the file. Right click it and choose Properties. The creation time has indeed changed to the early morning of December 1, 1622.

23. Examining Object Data

If you need to see all properties a result object provides, you should probably add Select-Object * to it like this:

```
Get-Process -Id $pid | Select-Object *
```

You will find that a much more thorough way uses Format-Custom. With this approach, you can specify a depth, which will allow you to see nested object properties down to the depth you specified:

```
Get-Process -Id $pid | Format-Custom * -Depth 5
```

The result looks similar to this:

```
PS> Get-Process -Id $pid | Format-Custom * -Depth 5

class Process
{
  __NounName = Process
  Name = powershell_ise
  Handles = 696
  VM = 1092001792
  WS = 225071104
  PM = 183767040
  NPM = 77760
  Path = C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
  Company = Microsoft Corporation
  CPU = 192,5832345
  FileVersion = 6.3.9421.0 (fbl_srv2_ci_mgmt_rel.130612-0600)
  ProductVersion = 6.3.9421.0
  Description = Windows PowerShell ISE
  Product = Microsoft® Windows® Operating System
  Id = 5976
  PriorityClass = Normal
  HandleCount = 696
  WorkingSet = 225071104
  PagedMemorySize = 183767040
  PrivateMemorySize = 183767040
  VirtualMemorySize = 1092001792
  TotalProcessorTime =
    class TimeSpan
    {
      Ticks = 1925832345
      Days = 0
      Hours = 0
      Milliseconds = 583
      Minutes = 3
      Seconds = 12
```

```
      TotalDays = 0,00222897262152778
      TotalHours = 0,0534953429166667
      TotalMilliseconds = 192583,2345
      TotalMinutes = 3,209720575
      TotalSeconds = 192,5832345
    }
  BasePriority = 8
  ExitCode =
  HasExited = False
  ExitTime =
  Handle = 1996
  MachineName = .
  MainWindowHandle = 2626616
  MainWindowTitle = Windows PowerShell ISE
  MainModule =
    class ProcessModule
    {
      ModuleName = PowerShell_ISE.exe
      FileName = C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
      BaseAddress = 5367201792
      ModuleMemorySize = 270336
      EntryPointAddress = 0
      FileVersionInfo =
        class FileVersionInfo
        {
          Comments =
          CompanyName = Microsoft Corporation
          FileBuildPart = 9421
          FileDescription = Windows PowerShell ISE
          FileMajorPart = 6
          FileMinorPart = 3
          FileName =
          C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
          FilePrivatePart = 0
          FileVersion = 6.3.9421.0 (fbl_srv2_ci_mgmt_rel.130612-0600)
          InternalName = POWERSHELL_ISE
(...)
```

24. Converting to Hex

Here's a simple way to convert a decimal to a hex representation, for example, if you want to display an error number in standard hexadecimal format:

```
PS> (-2147217407).ToString("X")
80041001
```

Use a lower-case "x" if you want the hex value to use lower-case letters.

```
PS> (212).ToString('x')
d4
```

To convert a hex value to a decimal, prepend it with "0x":

```
PS> 0x8004101
134234369
```

As you'll see, this conversion will assume unsigned values, so values are always positive.

You can also use the -f operator:

```
PS> "{0:x}" -f 12345678
bc614e
PS> "{0:X}" -f 12345678
BC614E
```

25. Creating New Objects

Most objects are returned by cmdlets and other commands, but you can also create new objects manually. Just use New-Object and specify the type of object you want.

This creates a new XML object, and you can then use its method Load() do open XML files or RSS feeds. This will load the RSS feed from powershellmagazine.com and display the top nodes:

```
$url = 'http://www.powershellmagazine.com/feed/'

$xmlresult = New-Object -TypeName XML

$xmlresult.Load($url)
$xmlresult
```

You can then traverse the nodes, so this gets you a list of all RSS feed entries:

```
$url = 'http://www.powershellmagazine.com/feed/'

$xmlresult = New-Object -TypeName XML

$xmlresult.Load($url)
$xmlresult.rss.channel.item |
  Select-Object -Property Title, pubDate, Link |
  Out-GridView
```

The names of the particular nodes (rss.channel.item in this case) depend on the XML document, of course.

In PowerShell 3.0, you can even use Out-GridView as a selection dialog, so this code would allow you to click on a feed item and then click OK to open in it your browser:

```
$url = 'http://www.powershellmagazine.com/feed/'

$xmlresult = New-Object -TypeName XML

$xmlresult.Load($url)
$xmlresult.rss.channel.item |
  Select-Object -Property Title, pubDate, Link |
  Out-GridView -PassThru -Title 'Select Topic' |
  ForEach-Object { Start-Process -FilePath $_.Link }
```

26. Using Constructors to Create New Objects

Some objects cannot be created using New-Object unless you specify additional initialization information. For example, if you tried to create a PSCredential object, you get an error message:

```
PS> $cred = New-Object -TypeName System.Management.Automation.PSCredential
New-Object : A constructor was not found. Cannot find an appropriate
constructor for type System.Management.Automation.PSCredential.
```

To list all constructors for the particular type you are trying to create, try this:

```
[System.Management.Automation.PSCredential].GetConstructors() |
  ForEach-Object {
        ($_.GetParameters() |
        ForEach-Object {
                '{0} {1}' -f $_.Name, $_.ParameterType.FullName
          }) -join ','
  }
```

This lists the constructors you can use with the type System.Management.Automation.PSCredential:

```
userName System.String,password System.Security.SecureString
pso System.Management.Automation.PSObject
```

So there are two constructors, and the first requires two arguments: a Username (type: string), and a password (type: SecureString).

So this is how you can automatically create a credential object that you then can present to any -Credential parameter to authenticate:

```
$username = 'test\user'
$password = 'topSecret' | ConvertTo-SecureString -Force -AsPlainText

$cred = New-Object -TypeName System.Management.Automation.PSCredential($username, $password)

$cred
```

Note how $password is converted into a SecureString type: since conversion of plain text to SecureString is not allowed by plain type conversions for security reasons, you can resort to ConvertTo-SecureString in this case.

Here's another piece of background information that you do not need to know, but it might help:

When you create the new object using New-Object, and additional parameters are required to create the new object, then you are not required to use parenthesis. This line would work as well:

```
$cred = New-Object -TypeName System.Management.Automation.PSCredential $username, $password
```

Parentheses indicate, though, that you are actually calling a method with parameters. Creating a new object always calls the internal "constructor" method (called "ctor"), and constructor methods may require additional parameters (like in this example). So, using parentheses helps create consistent code.

27. Displaying All Object Properties

Here is a fun example that uses New-Object to create a window with a property grid. You can use this to visualize objects and open a window that lists all object properties and values. Note however that bolded properties are writeable, so if you make changes in the property grid, this might change the underlying object you are viewing.

```
Function Show-Object
{
    param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [Object]
        $InputObject,

        $Title
    )

    if (!$Title) { $Title = "$InputObject" }
    $Form = New-Object System.Windows.Forms.Form
    $Form.Size = New-Object System.Drawing.Size @(600,600)
    $PropertyGrid = New-Object System.Windows.Forms.PropertyGrid
    $PropertyGrid.Dock = [System.Windows.Forms.DockStyle]::Fill
    $Form.Text = $Title
```

```
    $PropertyGrid.SelectedObject = $InputObject
    $PropertyGrid.PropertySort = 'Alphabetical'
    $Form.Controls.Add($PropertyGrid)
    $Form.TopMost = $true
    $null = $Form.ShowDialog()
}
```

Here are some examples how to use the new function:

```
Get-Process -Id $pid | Show-Object
$host | Show-Object
Get-Item -Path $pshome\powershell.exe | Show-Object
```

28. Using COM Objects

COM objects are not .NET objects. Instead, such objects use a much older technology. Still, COM objects are in wide use and can be helpful.

Older script languages like VBScript used COM objects a lot, and PowerShell has a cmdlet that works almost like VBScript's CreateObject: New-Object -ComObject.

WIA.ImageFile, for example, can return all kinds of useful information about images. The following lines show how to load an image file and display information like image size:

```
$image = New-Object -ComObject WIA.ImageFile
$image.LoadFile('C:\Users\Tobias\Pictures\img_0928.jpg')
$image
```

29. Listing Windows Updates

There is a not widely known COM object that you can use to list all the installed Windows Updates on a machine. Here is the code:

```
$Session = New-Object -ComObject Microsoft.Update.Session
$Searcher = $Session.CreateUpdateSearcher()
$HistoryCount = $Searcher.GetTotalHistoryCount()
if ( $HistoryCount -gt 0)
{
    $Searcher.QueryHistory(1,$HistoryCount) |
      Select-Object Date, Title, Description
}
```

30. Controlling Automatic Updates

To control whether Windows download and/or installs updates silently or prompts for permission, use this script and set the appropriate NotificationLevel via script.

Just make sure you run this code with full Administrator privileges. This code will set Automatic Downloads Notifications to level 3, so Windows will prompt you before it actually installs updates:

```
# run with full Admin privileges!

$updateObj = New-Object -ComObject Microsoft.Update.AutoUpdate
# ' 1 =  Never Check for Updates
# ' 2 =  Prompt for Update and Prompt for Installation
# ' 3 =  Prompt for Update and Prompt for Installation
# ' 4 =  Install automatically
$updateObj.Settings.NotificationLevel = 3
$updateObj.Settings.Save()
```

31. Controlling Automatic Updates Installation Time

To find out when Automatic Updates wakes your PC to install new updates, here is a script that will retrieve that information:

```
$updateObj = New-Object -ComObject Microsoft.Update.AutoUpdate
$day = $updateObj.Settings.ScheduledInstallationDay
$hour = $updateObj.Settings.ScheduledInstallationTime
$level = $updateObj.Settings.NotificationLevel

if ($level -eq 4) {
  if ($day -eq 0) {
    $weekday = 'every day'
  } else {
    $weekday = [System.DayOfWeek]($day-1)
  }

  "Automatic updates installed $weekday at $hour o'clock."
} else {
  'Updates will not be installed automatically. Check update settings for more info.'
}
```

To double-check settings or change them via UI, open the appropriate control like this:

```
$updateObj = New-Object -ComObject Microsoft.Update.AutoUpdate
$updateObj.ShowSettingsDialog()
```

32. Opening MsgBoxes

Need a quick message box to display something or ask a question? Fortunately, PowerShell can access old COM components. Here's a line that creates a MsgBox for 5 seconds. If the user does not make a choice within that time, it returns -1: a perfect solution for scripts that need to run unattended if no one is around.

```
$msg = New-Object -ComObject WScript.Shell
$msg.Popup("Hello", 5, "Title", 48)
```

To find out more about the Popup() method and its arguments, visit:

http://msdn.microsoft.com/en-us/library/x83z1d9f(v=VS.84).aspx

This link documents all the other WSH scripting methods as well.

33. Creating Custom Objects

PowerShell 2.0 has a new trick for creating your very own objects that you can then pass to other cmdlets. It is a two-step process. First, create a hash table and add all the information you want in your object:

```
$hash = @{}
$hash.Name = "Tobias"
$hash.Age = 85
$hash.hasDog = $true
```

Next, use this line to convert the hash table into an object:

```
$object = New-Object PSObject -Property $hash
$object
```

34. Creating Custom Objects with Select-Object

One very simple way of creating objects uses Select-Object:

```
$myObject = 'dummy' | Select-Object -Property Name, ID, Address
```

This gets you a new object in $myObject, and you can now fill in the values:

```
$myObject.Name = $env:username
$myObject.ID = 12
$myObject
```

35. Creating New Objects the JSON Way

There are numerous ways how you can create new objects that you may use to return results from your functions.

One way is using JSON, a very simple description language. It is fully supported in PowerShell 3.0. Have a look:

```
$data = '{"LastName":"Weltner","FirstName":"Tobias","Id":123}'
$myObject = ConvertFrom-Json $data
$myObject
```

36. Creating Objects in PowerShell 3.0 (Fast and Easy)

In PowerShell 3.0, you can cast a hash table to a PSCustomObject type to easily generate your own objects:

```
$data = @{
    LastName='Weltner'
    FirstName='Tobias'
    Id=123
}

$myObject = [PSCustomObject]$data
$myObject
```

Note that in PowerShell 2.0, conversion to PSCustomObject fails and you get back a hash table.

37. Discover Hidden Object Members

Get-Member is a great cmdlet to discover object members, but it will not show everything:

```
"Hello" | Get-Member
```

You should add -Force to really see a complete list of object members:

```
"Hello" | Get-Member -Force
```

One of the more interesting hidden members is called PSTypeNames and lists the types this object was derived from:

```
"Hello".PSTypeNames
```

38. Renaming Object Properties in PowerShell

Let's say you want to output just your top-level processes like this:

```
Get-Process |
Where-Object { $_.MainWindowTitle } |
Select-Object Name, Product, ID, MainWindowTitle
```

This works like a charm, but you'd like to rename the column "MainWindowTitle" to "Title" only. That's what AliasProperties are for:

```
Get-Process |
Where-Object { $_.MainWindowTitle } |
Add-Member -MemberType AliasProperty -Name Title -Value MainWindowTitle -PassThru |
Select-Object Name, Product, ID, Title
```

39. Getting Help for Objects - Online

In PowerShell 3.0, you finally can extend object types dynamically without having to write and import .ps1xml files. Here is an especially useful example:

```
$code = {
    $url = 'http://msdn.microsoft.com/en-US/library/{0}(v=vs.80).aspx' -f $this.GetType().FullName
    Start-Process $url
}

Update-TypeData -MemberType ScriptMethod -MemberName GetHelp -Value $code -TypeName System.Object
```

Once you execute this code, every single object has a new method called GetHelp(), and when you call it, your browser will open and show the MSDN documentation page for it - provided the object you examined was created by Microsoft, of course. There are many ways how you can call GetHelp(), for example:

```
$thedate = Get-Date

$thedate.GetHelp()
(Get-Date).GetHelp()
```

40. Finding Useful .NET Types

There are thousands of .NET types and no ultimate list of those that can be useful in PowerShell. However, the PowerShell team hard-coded a list of .NET types into PowerShell that they thought were useful.

Here's how to get to that list:

```
$typename = 'System.Management.Automation.TypeAccelerators'
$shortcut = [PSObject].Assembly.GetType($typename)::Get
$shortcut.Keys |
    Sort-Object |
    ForEach-Object { "[$_]" }
```

This gets you a list like this:

```
[adsi]
[adsisearcher]
[Alias]
[AllowEmptyCollection]
[AllowEmptyString]
[AllowNull]
[array]
[bigint]
[bool]
[byte]
[char]
[cimclass]
[cimconverter]
[ciminstance]
[cimtype]
[CmdletBinding]
[cultureinfo]
[datetime]
[decimal]
[double]
[float]
[guid]
[hashtable]
```

```
[initialsessionstate]
[int]
[int16]
[int32]
[int64]
(...)
```

41. Checking Loaded Assemblies

Use this line to check which .NET assemblies are currently loaded into PowerShell:

```
[AppDomain]::CurrentDomain.GetAssemblies()
```

42. Finding Methods with Specific Keywords

Use the next lines to find all .NET methods with a given keyword. In this example, you will get all type names that have a method with "Dialog" in its name:

```
$Keyword = 'Dialog'

$AllAssemblies = [AppDomain]::CurrentDomain.GetAssemblies()
$count = $AllAssemblies.Count
$i = 0

Foreach ($Assembly in $AllAssemblies)
{
    $percent = $i * 100 / $count
    $i++
    Write-Progress -Activity "Searching for method like '$Keyword'..." -Status $Assembly.FullName
-PercentComplete $percent

    $Types = $null
    try
    {
        $Types = $Assembly.GetExportedTypes()
    }
    catch
    {}
    $Types |
        Where-Object { $_.isPublic -and $_.isClass } |
        Where-Object { @($_.GetMethods() | Where-Object { $_.Name -like "*$Keyword*" }).Count -gt 0
} | Select-Object -ExpandProperty FullName
}
```