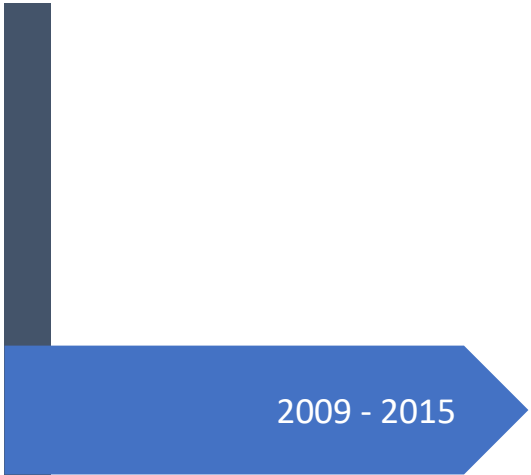
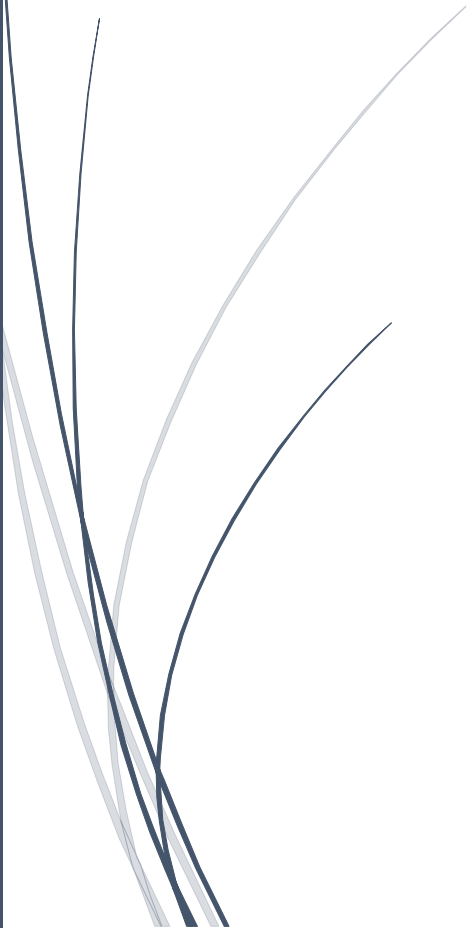


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the text "2009 - 2015".

2009 - 2015

Seriál PowerShell cz/sk technet blog

Several thin, curved lines in shades of blue and grey originate from the bottom left and sweep upwards and to the right.

Zdrojem je archiv českého a slovenského technet blogu:
<https://blogs.technet.microsoft.com/technetczsk>

Autorem textů je David Moravec

Obsah

Windows Powershell – úvod	9
1.1 Instalace	9
1.2 První krůčky	9
1.3 Základní pojmy	11
1.3.1 Cmdlet	11
1.3.2 Jmenná konvence	11
1.4 Bezpečnost	11
1.5 Náповěda	13
objekty a roury	14
Důležité vlastnosti	14
Objekty	14
Přístup k vlastnostem a metodám objektů	16
Roura	17
Zjištění vlastností a metod objektu	18
roury a aliasy	20
Práce s objekty v rouře	20
ForEach-Object a Where-Object	20
Select-Object	20
Sort-Object	21
Measure-Object	21
Group-Object	21
Compare-Object	21
New-Object, Tee-Object	22
Alias	22
dolujeme data aneb jak na WMI	23
1.1 Jak získat data z WMI	23
1.2 Parametry cmdletu Get-WmiObject	24
1.3 Parametr AsJob	25
souborový systém a registry	26
Základy souborového systému	27
1.3 Práce s ACL	28
Registr	28
Další PSDrives	29
PS pro programátory	29
Automatické zpracování	30

Práce s XML	31
Regex tester.....	32
Operátor -match.....	33
Akcelerátor [regex].....	33
Vytvoření objektu	34
Metoda FindR	35
Kódování, dekodování, konverze,	36
Převod Html2Xml	37
Clipboard	40
Je to všechno?	41
PS a Active Directory	42
1.1 Možnosti	42
1.2 Základní práce s AD	42
1.3 Vytvoření nového uživatele.....	44
1.4 Práce se skupinami	44
1.5 Další informace	44
tipy a triky.....	45
Konzole	45
Libovolná jména funkcí.....	45
Zápis bytů	46
Operátory a proměnné.....	47
Řetězení operátorů.....	47
Automatické proměnné \$\$, \$^.....	48
Objekty	48
Objekty a typy.....	48
Přidávání properties k PSObject.....	50
Práce s hashtable.....	50
Cmdlety	51
Splatting.....	51
Alias = Get-Alias.....	52
Vyhodnocení proměnné/výrazu v řetězci.	52
Parametr -OutputVariable.....	53
Ostatní Krátce	54
Nadstavby PowerShellu.....	55
PowerGUI	55
PSCX.....	57

SQLPSX.....	57
PowerShell Pack – WPK	58
PrimalForms Community Edition	58
ISE pack.....	59
Některé zajímavé zdroje.....	59
Pasti, chytáky a nechtěná překvapení.....	60
Funkce	67
Základní konstrukce při práci s funkcemi	67
Pokročilé funkce	69
Tvorba nápovědy.....	72
• Opravdu pokročilé funkce	74
• Použití [CmdletBinding()]	74
• Chcete opravdu zmáčknout to červené tlačítko?.....	76
• Použití WhatIf při každém volání.....	77
Skripty a moduly.....	79
Skripty	79
Profil	80
Moduly	81
Tipy a triky – mapy	83
Kostra.....	83
Závislosti mezi assemblies.....	84
Vytvoření kontrolek.....	85
Přidání křížku	87
Nastavení počátečních souřadnic.....	88
Povedlo se?.....	88
Download	88
Moduly	89
Psake.....	92
Proč psake a ne msbuild?	92
Jak psake rozchodit	92
Jednoduchý příklad.....	92
Parametrizace.....	94
K čemu jsou Parameters.....	95
Psake pro vývojáře	96
Psake do GUI.....	97
Změňte psake	98

Moduly – dokončení.....	99
Manifest	99
Závěrem.....	103
prohledáváme text	105
Základní porovnávání textu	105
Select-String.....	108
Závěrem.....	110
PowerShell z pohledu programátora.....	111
Framework	112
Typy	112
Bool.....	113
Řetězec	113
Scriptblock	113
Hashtable.....	114
Array	115
Dynamičnost.....	116
Operátory	116
Aritmetické	116
Logické	117
Přiřazení.....	117
Porovnání	118
Řetězcové operátory	118
Operátory na polích (kolekcích)	119
Funkce	119
Deklarace.....	119
Volání.....	120
Pipeline	121
OOP.....	123
Generiky	123
Metody	124
Scope, platnost proměnných.....	124
Vyhodnocování výrazů	126
Závěrem.....	128
Regulární výrazy a PowerShell.....	128
Operátor –match	128
Novinky Powershell verze 3	133

PowerShell v3	133
PowerShell Deep Dive	135
Představení PSCX.....	137
Import modulu	137
Ternární operátor a podobné pomůcky	139
Formátování	141
CD	143
Blbinka	144
Pro dnes vše	145
Get-LoremIpsum.....	145
Invoke-BatchFile	145
ConvertFrom-Base64, ConvertTo-Base64	145
Resolve-WindowsError, Resolve-HResult.....	146
Test-Xml, Format-Xml.....	147
echoargs	147
Get-FileVersionInfo	148
Help	149
Select-String	149
Základní použití Select-String	150
Prohledávání logů.....	152
Prohledávání více souborů	154
Microsoft Script Explorer for Windows PowerShell.....	155
Operátor formátování	162
Formátování ve formátování.....	164
Tvorba vlastních objektů	168
Typ objektu.....	171
PowerShell v3	174
\$PSDefaultParameterValues	175
Update-Help	175
Nové možnosti vytváření vlastních objektů	176
PowerShell v3 – další zajímavé novinky	177
Restart-Computer.....	177
PowerShell Web Access.....	177
jak na sdílené složky	181
Automatické načítání modulů	181
Sdílené složky	181

Get-SmbShare.....	182
New-SmbShare	183
Set-SmbShare	184
Remove-SmbShare	185
*-SmbOpenFile	185
Další zajímavé funkce	186
práce se síťovým adaptérem.....	188
Modul NetTCPIP	188
Get-NetIPConfiguration.....	189
Get-NetIPInterface	190
NetAdapter	191
Get-NetAdapter	194
vylepšená práce s WMI	194
WMI vs. CIM	195
jak na webové stránky.....	198
Invoke-WebRequest	198
přechod z příkazové řádky.....	201
Active Directory	202
Ping, ipconfig, nslookup.....	205
PowerShell a Azure Active Directory	206
Jak se připojit do Azure Active Directory (AAD)	206
Závěrem.....	209
PowerShell v4	209
Malé změny v cmdletech.....	210
Desired State Configuration (DSC)	213
Hrátky s Hyper-V.....	216
Tip na práci s časem	219
Transakce.....	221
Vylepšený Where-Object.....	225
Vylepšená konzole	228
Zvýrazňování syntaxe.....	228
Doplňování	228
Tipy	230
Cyklotoulky	230
For.....	230
While	231

Do	231
Foreach.....	231
Závěrem.....	232
Záchranné parametry	233
Novinky verze 5	234
E+T v Praze.....	234
Aprílová MVP nominace	235
Novinky: WMF 5 Preview	235
//build/	238
Přejímání skriptů z internetu	238
Script Browser & Script Analyzer.....	242
Vyhledávání skriptů	242
Kontrola skriptů.....	243
Krátké tipy na prázdniny.....	245
Převod mezi megy a kily	245
OGV	246
ISESteroids 2.0.....	247
CopyFrom-Help – kopírování příkladů z nápovědy do ISE.....	247
Show-Reference – odkazy na použití daného kódu	248
Invoke-Versioning + Show-Diff – verzování dokumentu přímo z ISE	248
Invoke-ExternalConsole – ISE konzole jako undocked okno	249
Invoke-Fix – spuštění „oprav“ kódu	249
Select-MultiText – označení stejného textu napříč dokumentem	249
Split View – rozdělení okna	250
Oficiální uvedení.....	250
Vzdálený přístup poprvé	250
Ideální situace – doména	251
Nastavení práv pro vzdálený přístup.....	253
Zpracování textu.....	254
Zpracování textu, vNext	259
Vánoce	260
PowerShell a Azure REST API.....	260
Příklady.....	261
Tip	264
Hash tabulka.....	265
Jak hash tabulka vypadá?	265

Použití.....	266
Export do HTML.....	268
PowerShell v5	271
Transcript.....	271
Symbolic links	271
Práce s archivy.....	271
Zpracování nestrukturovaného textu.....	271
OneGet a PowerShellGet.....	271
PSScriptAnalyzer.....	272
Script Debug	272
PSScriptAnalyzer.....	273
Desired State Configuration	273
Cryptographic Message Syntax	274
Nano Server	274
Windows 10 a PowerShell v5	278
Konzole	278
Zvětšení okna.....	278
Vybrání bloku textu	278
Výběr textu pomocí klávesnice	279
Další zajímavosti	280
Vsuvka.....	280

WINDOWS POWERSHELL – ÚVOD

PowerShell je skriptovací jazyk a shell od Microsoftu. Z předchozí věty jsem záměrně vypustil slovo nový, protože první verze byla vydána již v roce 2006. Verze 2 je v současné době ve stadiu CTP3 (Community Technology Preview) a její finální uvedení je plánováno současně s vydáním Windows 7.

Tato série si klade za cíl seznámit vás s PowerShellem od úplných základů až po složitější konstrukce. Nemá být (ani být nemůže) vyčerpávajícím zdrojem. Pokud se budete chtít PowerShell naučit, budete muset pokračovat dále, např. studiem nápovědy, která je až překvapivě dobrá a užitečná. Začneme instalací.

1.1 Instalace

PowerShell je součástí posledních verzí operačních systémů, tedy Windows 7 a Windows 2008 R2. Ve Windows 2008 je dostupný jako volitelný doplněk a můžete jej doinstalovat přes Server Manager. Pokud jej budete chtít použít ve Windows Vista, XP a 2003 Server, musíte si jej doinstalovat jako [samostatné KB](#). Pro nižší verze není PowerShell dostupný. V současné době je možno PowerShell doinstalovat i přes Windows Update (nebo WSUS).

Před instalací PowerShellu se ujistěte, že máte nainstalován .NET Framework minimálně ve verzi 2.0. Výše zmiňované KB pak nainstalujete klasickým next, next, next způsobem. Po instalaci naleznete v nabídce *Start* složku *Windows PowerShell 1.0* a v ní modrou ikonu, kterou si určitě zamilujete J

1.2 První klíčky

Po spuštění uvidíte následující modrou obrazovku – tato modrá (#012456) je dobrá.

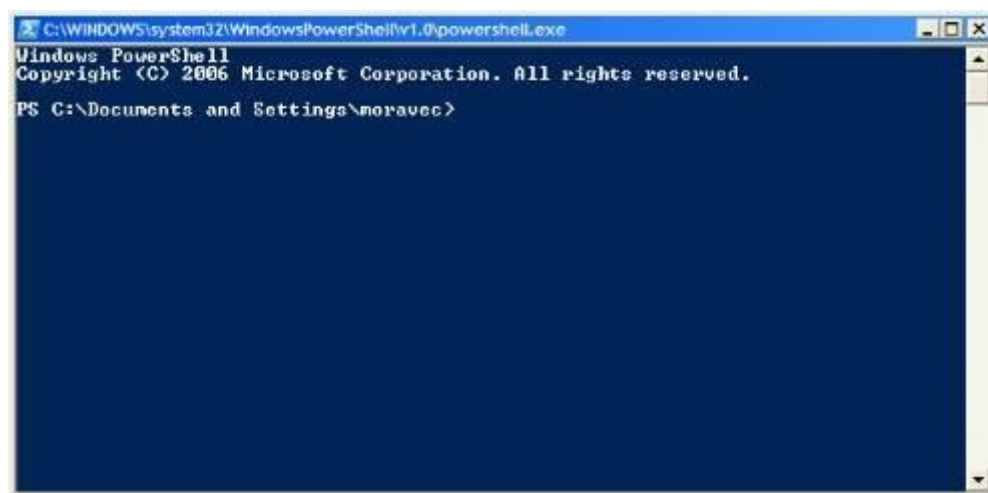


Figure 1: Okno PowerShellu

Zkusme si ukázat první obligátní kus kódu – známý Hello World! V PowerShellu jej zapíšete následujícím jednoduchým příkazem:

```
PS C:\> "Hello World!"
```

PowerShell vezme váš vstup a text opíše na výstupu. Mimochodem, pěkný přehled Hello World programů můžete najít na [Wikipedii](#). Pojdme si ukázat několik užitečnějších příkazů, např. pro zobrazení aktuálního data a času

```
PS C:\> Get-Date
```

```
3. června 2009 22:00:32
```

pracovního adresáře

```
PS C:\> Get-Location
```

```
Path
```

```
----
```

```
C:\
```

nebo zjištění nastavení jazykového prostředí ve Windows.

```
PS C:\> Get-Culture
```

```
LCID Name DisplayName
```

```
-----
```

```
1029 cs-CZ Czech (Czech Republic)
```

Chtěli byste například zpracovat v PowerShellu výsledky z vyhledávače [Bing](#)?

```
PS C:\> Get-BingWeb 'technet powershell site:cz' | Format-Table Url
```

```
Url
```

```
---
```

```
http://www.logon.cz/?p=129
```

```
http://www.wug.cz/Default.aspx?tabid=53&EntryID=75
```

```
http://www.logon.cz/?tag=powershell
```

```
http://www.wug.cz/Default.aspx?tabid=58&mid=500&ctl=Detail&ItemID=208&VisibleDate=1.12.2008
```

```
http://blog.vyvojar.cz/dotnet/archive/2006/11/30/138593.aspx
```

```
http://www.wug.cz/Default.aspx?tabid=53&EntryID=106
```

```
http://www.wug.cz/Aktuality/tabid/36/ctl/Detail/mid/492/ItemId/270/language/cs-CZ/Default.aspx
```

```
http://www.logon.cz/?p=230
```

```
http://www.wug.cz/Default.aspx?tabid=58&mid=500&ctl=Detail&ItemID=144&VisibleDate=9.5.2008
```

```
http://www.cs.vsb.cz/navrat/vyuka/sps/prednasky/pred10b.pdf
```

Příkaz vyhledá text „TechNet Powershell“ v českých doménách a pomocí **Format-Table** vypíše adresy prvních deseti odkazů, které nalezne. **Get-BingWeb** není standardním příkazem PowerShellu, ale je na něm ukázána jedna ze silných stránek PowerShellu – komunita. Již druhý den po ohlášení nového vyhledávače vytvořil [Joe Pruitt](#) knihovnu funkcí pro práci s bingem (PoshBing) a uvolnil ji pro veřejné použití. Pokud ji chcete také vyzkoušet, je ke stažení na [CodePlexu](#).

1.3 Základní pojmy

Než se pustíme do podrobnějšího přehledu, řekneme si něco o terminologii v PowerShellu. Jedná se o základní poučky, které vám umožní začít PowerShell efektivněji a rychleji používat.

1.3.1 Cmdlet

Je základním příkazem PowerShellu. Vyslovuje se jako *command-let*. Ve verzi 1 je jich v PowerShellu 129. V příkladech výše jsme si ukázali cmdlety **Get-Date**, **Get-Location** a **Get-Culture**.

1.3.2 Jmenná konvence

V předchozím bodě jsme si ukázali tři cmdlety. Nezdá se vám na jejich jméně něco zajímavého? Pokud vám přijde zajímavá konvence **Get-<něco>**, jste na správné stopě. Vývojový tým PowerShellu zavedl jmennou konvenci, která je v angličtině vyjádřena jako **Verb-Noun**, čili v češtině **Sloveso-PodstatnéJméno**. Zároveň se snaží (a dle mých zkušeností úspěšně) tuto konvenci prosadit. V současné době existuje malá skupina sloves pro cmdlety a je v hojně míře používána. [Více](#) se můžete dozvědět přímo v [blogu vývojového týmu PowerShellu](#).

Seznam sloves ve verzi 1 je následující:

Add	Clear	Compare	ConvertFrom	Convert
ConvertTo	Copy	Export	ForEach	Format
Get	Group	Import	Invoke	Join
Measure	Move	New	Out	Pop
Push	Read	Remove	Rename	Resolve
Restart	Resume	Select	Set	Sort
Split	Start	Stop	Suspend	Tee
Test	Trace	Update	Where	Write

Pro podstatné jméno platí pravidlo, že musí být v jednotném čísle. Proto existuje cmdlet **Get-QADUser** a nikoli **Get-QADUsers**.

Velkou výhodou této snahy je možnost „odhadnout“ jméno cmdletu, který právě potřebujete. Z příkladů je to víceméně jasné: chcete-li zjistit aktuální datum, použijte **Get-Date**; chcete-li datum nastavit, použijte **Set-Date**.

Na základě předchozích řádek asi dokážete dát dohromady příkaz, kterým zjistíte běžící procesy – je to **Get-Process**.

1.4 Bezpečnost

Řekněme si ještě něco krátce k bezpečnosti PowerShellu. Microsoft se poučil z hrozeb, které ohrožovaly počítače pomocí skriptů ve VBS a jedním z kroků je, že skripty v PowerShellu (přípona PS1) jsou po nainstalování asociovány s poznámkovým blokem. Tím odpadá „náhodné“ spuštění destruktivního kódu pouhým dvojklikem.

Několik dalších bezpečnostních prvků si ukážeme na příkladu. Zkuste vytvořit textový soubor, který bude obsahovat pouze jednu řádku textu, na které bude zapsán cmdlet Get-Date. Uložte tento soubor

jako skript.ps1. Spustíte PowerShell a přejděte v něm do adresáře s vytvořeným skriptem. Pro pohyb použijte příkaz **cd**, známý z cmd.exe. Následujícím příkazem zkuste skript spustit:

```
PS C:\Temp\PS> skript.ps1
```

```
The term 'skript.ps1' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.
```

```
At line:1 char:10
```

```
+ skript.ps1 <<<<
```

Narazili jste na další „bezpečnostní překážku“ – PowerShell vám neumožní spustit skripty z aktuálního adresáře, pokud napíšete pouze jejich jméno. Příkaz musíte napsat následujícím způsobem:

```
PS C:\Temp\PS> ./skript.ps1
```

```
File C:\Temp\PS\script.ps1 cannot be loaded because the execution of scripts is disabled on this system. Please see "get-help about_signing" for more details.
```

```
At line:1 char:2
```

```
+ & <<<< "C:\Temp\PS\script.ps1"
```

Skript se vám opět nespustil, ale chybové hlášení je jiné. Dalším krokem k zabezpečení počítače jsou bezpečnostní politiky (v angličtině Execution policies), které hlídají, zda jsou spouštěné skripty digitálně podepsány. Po nainstalování je nastavena politika *Restricted*. V tomto módu je umožněno zapisování příkazů přímo v okně PowerShellu, ale pokud chcete spustit skript, PowerShell vám v tom zabrání a zobrazí chybu.

Bezpečnostní politiky jsou celkem čtyři: *Restricted*, *AllSigned*, *RemoteSigned*, *Unrestricted*. Pokud ji chcete změnit, použijte příkaz **Set-ExecutionPolicy** a jako parametr použijte jméno jedné ze jmenovaných politik.

```
Set-ExecutionPolicy RemoteSigned
```

zajistí, že skripty stažené z internetu nebo doručené mailem budou spuštěny, ale pouze pokud jsou podepsány důvěryhodným certifikátem. Nyní už bez problému spustíte skript, který jste vytvořili.

```
PS C:\Temp\PS> ./skript.ps1
```

```
3. června 2009 23:31:26
```

Pokud se chcete o politikách dozvědět více, zkuste v PowerShellu spustit příkaz

`Get-Help about_signing`

Je vidět, že Microsoft se poučil z minulosti a již od začátku navrhl PowerShell i s ohledem na bezpečnost. Poslední krok je ale samozřejmě vždy na uživateli. Pokud změníte bezpečnostní politiku na Unrestricted, není chybou PowerShellu, že spustíte skript z neznámého zdroje a poškodíte svůj počítač, či celou síť.

Doporučení: Nechte na svých počítačích politiku nastavenou alespoň na RemoteSigned, vyhnete se tak mnoha případným problémům.

1.5 Náповěda

Velmi silnou stránkou PowerShellu je jeho nápověda. Již při instalaci se nainstalují tři zajímavé dokumenty: User Guide, Getting Started a Quick Reference. Dohromady se jedná asi o 150 stran doporučeného čtení.

Další možností je nápověda vestavěná přímo uvnitř shellu, dostupná je pomocí příkazu **Get-Help**. Jako administrátoři máte možná snahu nápovědu zatracovat, podceňovat či vůbec tak nějak nenávidět. V tomto případě by to byla chyba. Nápověda je zpracována velice dobře a už se mnohokrát osvědčila.

```
PS C:\> Get-Help Get-Date

NAME

    Get-Date

SYNOPSIS

    Gets the current date and time.

SYNTAX

    Get-Date [[-date] <DateTime>] [-displayHint {<Date> | <Time> | <DateTime>}] [-format <string>] [-year
<int>] [-month <int>] [-day <int>] [-hour <int>] [-minute <int>] [-second <int>] [<CommonParameters>]

    Get-Date [[-date] <DateTime>] [-displayHint {<Date> | <Time> | <DateTime>}] [-uFormat <string>] [-year
<int>] [-month <int>] [-day <int>] [-hour <int>] [-minute <int>] [-second <int>] [<CommonParameters>]

DETAILED DESCRIPTION

    Gets the current date and time.

RELATED LINKS

    Set-Date
    New-TimeSpan

REMARKS

    For more information, type: "get-help Get-Date -detailed".
    For technical information, type: "get-help Get-Date -full".
```

Toto je základní výpis informace o vybraném cmdletu. Všimněte si posledních dvou řádek a jejich doporučení. Nejpodrobnější informace získáte pomocí přepínače **-Full**. Součástí nápovědy ke každému cmdletu je ukázka příkladů použití. Pokud použijete přepínače **-Full** nebo **-Detailed**,

budou příklady zobrazeny na konci výpisu. Pokud chcete pouze výpis příkladů, použijte přepínač – **Examples**.

```
Get-Help Get-Date -Examples
```

V PowerShellu existuje ještě jeden typ nápovědy – tematická. Každé z témat popisuje použití některých částí PowerShellu, např. aritmetické operace, regulární výrazy, aliasy, atd. Seznam všech témat je zobrazen níže.

```
about_alias
about_assignment_operators
about_break
about_commonparameters
about_core_commands
about_escape_character
about_flow_control
about_function
about_if
about_logical_operator
about_object
about_parsing
about_property
about_quoting_rules
about_regular_expression
about_script_block
about_special_characters
about_types
about_wildcard
about_arithmetic_operators
about_associative_array
about_command_search
about_comparison_operators
about_display.xml
about_execution_environment
about_for
about_globbing
about_line_editing
about_method
about_operator
about_path_syntax
about_provider
about_redirection
about_reserved_words
about_shell_variable
about_switch
about_where
about_array
about_automatic_variables
about_command_syntax
about_continue
about_environment_variable
about_filter
about_foreach
about_history
about_location
about_namespace
about_parameter
about_pipeline
about_pssnapins
about_ref
about_scope
about_signing
about_system_state
about_while
```

Dnes jsme si ukázali některé základní vlastnosti PowerShellu. Příště si povíme o objektech, rouře a provázanosti PowerShellu s .NET Frameworkem. Těšte se...

OBJEKTY A ROURY

Důležité vlastnosti

Když někomu řeknete, aby vám vysvětlil výhody PowerShellu, většinou se dozvíte, že jednou z nich je objektová roura. Mechanismus roury už většina z vás asi použila (nebo jste o ní alespoň slyšeli), horší to už bude zřejmě s tou její objektivostí. Práce s objekty vychází z toho, že PowerShell je postavený na platformě .NET Framework (vzpomínáte si, když jsme si v první části, že bez .NETu PowerShell nenainstalujete?). V dnešní části si povíme právě o objektech a rouře. O .NETu si povíme příště.

Objekty

Začněme obecným vysvětlením principu objektů. Ti z Vás, kdo máte znalosti jakéhokoli objektového jazyka, můžete tuto část přeskočit. Rovnou se omlouvám některým .NET puristům – určité části budu záměrně zjednodušovat. Tento článek je psán z pohledu administrátora a (v tuto chvíli) není nutné vědět, co je například konstruktor.

Při svých prezentacích PowerShellu pro lidi zabývající se administrací systémů jsem zjistil, že některým z nich dělá problém pochopit základní princip. Ukazuji jim tedy analogii na lahvích různých tvarů a náplní. Při druhém vyprazdňování lásky s rumem je jim to již většinou jasné. Na stránky seriózního média, jakým bezesporu TechNet je, ovšem alkohol nepatří a tak se spokojíme s něčím méně kontroverzním. Upustíme od tradičních žárovek nebo aut a přeneseme si do objektů svět zvířat – budeme pracovat s objektem kočkovitá šelma.

Každý objekt si sebou nese sadu vlastností a metod (funkcí). V případě kočky mezi vlastnosti patří např. velikost, barva chlupů, barva očí, váha či oblíbené jídlo. Řeknu-li tedy, že máme kočku oranžové barvy s černými pruhy, váhou 250kg, velikostí XXXL, která pojídá s oblibou buvolu, bude vám jasné, že mluvím o tygrovci. Pokud taková kočka váží šest kilo, je tříbarevná, velikosti M, má ráda šunku a její oblíbené místo k válení je klávesnice mého notebooku, asi tušíte, kde byste takovou kočku hledali.

Stejně tak to je s objekty v počítačovém světě. Pokud se budeme bavit například o procesech jsou jejich vlastnostmi **ProcessName**, **WorkingSet**, **StartTime** nebo **Id**. Vlastnosti popisují určitý stav objektu v čase, pokud chcete takový stav změnit, použijete k tomu určené metody.

Kočka má například metody spát, jíst, značkovat, vrátit nebo plést se pod nohy. Pokud u kočky spustíte metodu spát, můžete se jít v klidu dívat na televizi (dokud nevyvoláte u kočky metodu probudit se, například otevřením dveří od ledničky). U procesů je to podobné (s tou výjimkou, že procesy našťestí na ledničku nereagují), zavoláním určité metody změníme chování nebo stav procesu. Existují například metody **Start()**, **Refresh()** nebo **Kill()**.

Všimněte si, že jsem za názvy metod psal kulaté závorky. Berte to jako mnemotechnickou pomůcku, pokud uvidíte při práci s objekty závorky, pracujete s metodou.

Objekty a objektový přístup obecně, mají jednu obrovskou výhodu. Relativně jednoduše s nimi dokážeme pracovat pomocí základních příkazů. Když jsme si šli vybírat do útulku objekt kočka, věděli jsme, že chceme určitou velikost, barvu, roztomilost a způsob chování (samozřejmě že to dopadlo úplně jinak, ale to nebyla chyba objektového přístupu, ale změny vstupních parametrů v okamžiku vstupu do místnosti plné mňoukajících koťátek). U procesů je to našťestí exaktnější. Počítač vám běží pomalu a tak hledáte process, který zabírá nejvíce procesorového času či paměti a na ten se zaměříte. „Zmrzla“ vám určitá aplikace a tak hledáte process s jejím jménem, abyste ho ukončili, resp. zavolali nad oním procesem metodu **Kill()**.

Některé z typů objektů (s jejich vlastnostmi a metodami) jsou vypsány v následující tabulce. Co je třída si řekneme na konci dnešního článku.

Jméno	Třída	Vlastnosti	Metody
Proces	System.Diagnostics.Process	ProcessName, WorkingSet, StartTime	Start(), Refresh(), Kill()
Služba	System.ServiceProcess.ServiceController	DisplayName, Status, DependentServices	Start(), Stop(), Pause()
Soubor	System.IO.FileInfo	Name, Length, LastAccessTime	Open(), Delete(), Delete()

Klíče registru	Microsoft.Win32.RegistryKey	Name, SubKeyCount, ValueCount	OpenSubkey(), GetValue(), DeleteValue()
Datum	System.DateTime	Date, Hour, DayOfWeek	Add(), AddDays(), ToUniversalTime()
Text	System.String	Length	CompareTo(), Contains(), ToLower()

Přístup k vlastnostem a metodám objektů

Než si ukážeme jak přistupovat k vlastnostem a metodám objektů hromadně pomocí roury, ukážeme si, jak k nim přistupovat v případě jednoho objektu. V následujících ukázkách si nejdříve uložíme textový řetězec do proměnné (\$text) a poté budeme přistupovat k jednotlivým vlastnostem a metodám.

```
PS C:\> $text = "Ahoj"
PS C:\> $text.Length
4
PS C:\> $text.ToLower()
ahoj
PS C:\> $text.ToUpper()
AH0J
PS C:\> $text.Replace("hoj", "nezka")
Anezka
PS C:\> $text.EndsWith("nezka")
False
```

Za chvíli si ukážeme, jak zjistit, které vlastnosti a metody lze pro konkrétní typ objektu použít.

Tip: PowerShell podporuje doplňování příkazů pomocí klávesy tabulátor <Tab>. Pokud byste v předchozím příkladu napsali \$te<Tab>.Le<Tab>, PowerShell zápis doplní na správnou formu (\$text.Length). Tabulátorem můžete „cyklovat“ mezi možnými variantami – v případě, že napíšete pouze \$text. (tečka je zde důležitá), můžete opakovaným použitím klávesy <Tab> procházet všechny dostupné vlastnosti a metody.

Další možností je doinstalovat nástroj [PowerTab](#), který po stisknutí <Tab> zobrazí dostupné vlastnosti a metody přehledněji.



Roura

Mechanismus roury (v angličtině *pipeline*) se ve výpočetní technice používá od sedmdesátých let, kdy byla tato technika implementována do operačního systému UNIX. Její idea je geniálně jednoduchá: „Vezmi výstup jednoho příkazu a pomocí roury jej pošli na vstup příkazu dalšího.“ Roura se zapisuje pomocí znaku `|` a byla dostupná již v dřívějších verzích operačních systémů Microsoftu. V `cmd.exe` (nebo již v MS-DOSu) jste ji mohli bez problémů použít např. pro stránkování dlouhých výpisů:

```
C:\>dir /b /s *.txt | more
```

Výstup příkazu `dir` (seznam txt souborů) je poslán na vstup příkazu `more`, který zajistí zastavení výpisu po zaplnění obrazovky. V PowerShellu můžete rouru použít třeba takto:

```
PS C:\> dir *.ps1 | Sort -Property Length
```

Na výstupu příkazu **dir** jsou objekty (kolekce objektů) a každý z nich reprezentuje jeden soubor s příponou PS1. Tyto objekty jsou pomocí roury předány na vstup příkazu **Sort**, který je vezme a seřadí je podle vlastnosti **Length**. Na výstupu příkazu **Sort** máme tedy soubory PS1 seřazené podle velikosti (opět se zde nacházejí ve formě kolekce objektů, takže bychom je mohli další rourou poslat jinému příkazu). PowerShell ví, že vlastnost **Length** je číselný údaj a že s ním má tedy jako s číslem pracovat. Stejně tak ví, že když napíšeme

```
PS C:\> dir *.ps1 | Sort -Property LastWriteTime
```

chceme řadit soubory podle časového údaje a sám provede potřebné úkony. Pokud tedy pracujeme s objekty v rouře, stačí vědět, kterou vlastnost potřebujeme pro dosažení potřebných výsledků.

Pojďme si ukázat ještě pár jednoduchých příkladů a potom si řekneme, jak vlastně zjistíme, které vlastnosti obsahuje konkrétní typ objektu.

Příkaz	Výsledek
<code>dir Sort -Property Length -Descending Select -First 5</code>	Pět největších souborů v adresáři
<code>dir Group -Property Extension Sort -Property Count -Descending</code>	Jednotlivé typy souborů, řazené podle počtu
<code>dir Measure-Object -Property Length -Sum</code>	Celková velikost souborů v adresáři
<code>Get-WmiObject Win32_LogicalDisk ` Select Name, DriveType, FileSystem, Size, FreeSpace   ` Format-Table -AutoSize</code>	Vypíše logické disky, jejich jméno, typ, souborový systém, velikost disku a volné místo. Výsledek zobrazí v tabulce.
<code>Get-Process -Name notepad Stop-Process</code>	Ukončení notepadu

U posledního příkladu je dobré si uvědomit jednu věc – PowerShell najde **všechny** procesy, které se jmenují notepad a pošle je rourou dále. **Stop-Process** převezme **opět všechny** procesy a zastaví je. Pokud byste se překlepli a provedli následující příkaz (nezkoušet!!!)

`Get-Process | Stop-Process`

způsobili byste svému počítači (s Windows XP) modrou smrt, protože byste zastavili všechny běžící procesy. Abyste takovým věcem zabránili, existují v PowerShellu příkazy a přepínače, které si ukážeme v některém z následujících článků.

[Zjištění vlastností a metod objektu](#)

Nyní si ukážeme, jak zjistit, které vlastnosti a metody obsahuje konkrétní objekt. V PowerShellu je k tomuto účelu cmdlet **Get-Member**. Bez přehánění můžu prohlásit, že to je nejužitečnější cmdlet, který existuje. Ukažme si jeho použití nejprve na jednoduchém řetězci znaků (výstup byl zkrácen).

```
PS C:\> "Ahoj" | Get-Member

TypeName: System.String

Name      MemberType      Definition
-----
Clone     Method          System.Object Clone()
CompareTo Method          System.Int32 CompareTo(Object value),
Contains  Method          System.Boolean Contains(String value)
CopyTo    Method          System.Void CopyTo(Int32 sourceIndex,
EndsWith  Method          System.Boolean EndsWith(String value)
Equals    Method          System.Boolean Equals(Object obj), Sy
...
Trim      Method          System.String Trim(Params Char[] trim
TrimEnd   Method          System.String TrimEnd(Params Char[] t
TrimStart Method          System.String TrimStart(Params Char[]
Chars     ParameterizedProperty System.Char Chars(Int32 index) {get;}
Length    Property        System.Int32 Length {get;}
```

Vidíte, že řetězec má jednu vlastnost (Length) a množství metod (ve skutečnosti je jich 35). Jak je použít jsme si ukázali před chvílí. Pojd' me si zobrazit vlastnosti objektu typu soubor.

```
PS C:\> dir config.sys | gm -MemberType Property

TypeName: System.IO.FileInfo

Name      MemberType Definition
-----
Attributes Property  System.IO.FileAttributes Attributes {get;set;}
CreationTime Property  System.DateTime CreationTime {get;set;}
CreationTimeUtc Property  System.DateTime CreationTimeUtc {get;set;}
Directory  Property  System.IO.DirectoryInfo Directory {get;}
DirectoryName Property  System.String DirectoryName {get;}
Exists      Property  System.Boolean Exists {get;}
Extension   Property  System.String Extension {get;}
FullName     Property  System.String FullName {get;}
IsReadOnly  Property  System.Boolean IsReadOnly {get;set;}
LastAccessTime Property  System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property  System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime Property  System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc Property  System.DateTime LastWriteTimeUtc {get;set;}
Length      Property  System.Int64 Length {get;}
Name        Property  System.String Name {get;}
```

Zde vidíte, jak jsem zjistil, že existuje vlastnost LastWriteTime.

Get-Member je opravdu vynikající průzkumník a pomocník. Zvláště ze začátku jej budete používat velice často.

Ještě jednu věc je potřeba zmínit. Všechny objekty stejného typu (např. procesy) jsou odvozeny od tzv. třídy. Třída říká, jaké vlastnosti a metody bude objekt mít. Nemusíte si pamatovat, že proces (např. notepad.exe) je vlastně odvozen od třídy **System.Diagnostics.Process**. Je ale dobré vědět, kde tuto informaci hledat pro případ, že byste chtěli např. na MSDN hledat podrobnější informace o objektu. Příkaz **Get-Member** vám tuto informaci zobrazí v první řádce výpisu.

Můžete také použít následující kus kódu (schválně jsem použil relativně složitý a nečitelný zápis – chci vás tím nalákat na další část, kde si povíme mimo jiné o aliasech):

```
PS C:\> dir | gm |% {$_.TypeName} | Group | ft Name

Name
----
System.IO.DirectoryInfo
System.IO.FileInfo
```

Příkaz `dir` můžete změnit za jakýkoli jiný, který vás zajímá. Chcete-li ověřit, že text „Ahoj“ je opravdu string, můžete použít

```
PS C:\> "Ahoj" | gm |% {$_.TypeName} | Group | ft Name

Name
----
System.String
```

ROURY A ALIASY

Práce s objekty v rouře

V předchozím díle jsme si ukázali jak lze jednoduše pracovat s objekty v rouře. Až doposud jsme pracovali s celou kolekcí objektů (např. všechny procesy, které nám PowerShell vrátil). Nyní si ukážeme, jak filtrovat objekty dle potřeby.

Pro práci s objekty slouží následující cmdlety:

```
C:\> Get-Command *-object | Format-Table Name
Name
----
Compare-Object
ForEach-Object
Group-Object
Measure-Object
New-Object
Select-Object
Sort-Object
Tee-Object
Where-Object
```

Na ty nejčastěji používané se nyní podíváme podrobněji.

ForEach-Object a Where-Object

Jsou určitě nejpoužívanějšími z „objektových“ cmdletů. **ForEach-Object** přijme všechny objekty z roury a provede nad nimi danou operaci. **Where-Object** přijme také všechny objekty z roury, ale k dalšímu zpracování pošle pouze ty, které splňují danou podmínku. Čili:

```
C:\> Get-WmiObject Win32_LogicalDisk | `
ForEach-Object { Write-Host $_.Name $_.Description $($_.FreeSpace / 1GB) }

C: Local Fixed Disk 3,89662170410156
D: CD-ROM Disc 0
```

Zpracuje z WMI všechny logické disky a vypíše jejich jméno, popis a prázdné místo v gigabytech. Pokud by nás zajímaly pouze pevné disky, mohli byste použít **Where-Object** pro odfiltrování nepotřebných záznamů:

```
C:\> Get-WmiObject Win32_LogicalDisk | `
Where-Object { $_.DriveType -eq 3 } | `
ForEach-Object { Write-Host $_.Name $_.Description $($_.FreeSpace / 1GB) }

C: Local Fixed Disk 3,89662170410156
D: CD-ROM Disc 0
```

Pomocí **Where-Object** jsme určili, že chceme pouze disky, které mají DriveType rovno tři (což lze [jednoduše vyhledat](#) v MSDN).

Pokud se vám nelíbí zobrazení výsledku, můžete použít [operátor formátování \(-f\)](#):

```
C:\> Get-WmiObject Win32_LogicalDisk | `
ForEach-Object { "{0,3} {1,-17} {2:F2} GB" -f $_.Name, $_.Description, $($_.FreeSpace / 1GB) }

C: Local Fixed Disk 3,90 GB
D: CD-ROM Disc 0,00 GB
```

Select-Object

Tento cmdlet slouží k definování vlastností, které chceme předávat dále rourou. Srovnejte například následující příklady:

```
PS C:\> Get-Process
PS C:\> Get-Process | Select-Object ProcessName, Id, Handles
```

Select-Object vezme z objektu uvedené vlastnosti a pošle je dále rourou. Tento cmdlet je užitečný hlavně při exportech, např. při použití **Export-Clixml**.

Poznámka: Možná si říkáte, proč nepoužít místo **Select-Object** např. **Format-Table**, když oba dávají stejné výsledky:

```
PS C:\> Get-Process | Format-Table ProcessName, Id, Handles
```

Slovo stejné bych měl dát do uvozovek. Rozdíl je v tom (zde se dopustím úmyslně zjednodušení), že **Select-Object** posílá rourou dál pořád objekty, kdežto **Format-Table** pouze zapisuje **textovou informaci (!)** na standardní výstup (zde konzole). Můžete si to vyzkoušet jednoduchým pokusem:

```
PS C:\> (Get-Process)[0..4] | Select-Object ProcessName, Id, Handles | Select-Object ProcessName
PS C:\> (Get-Process)[0..4] | Format-Table ProcessName, Id, Handles | Select-Object ProcessName
```

Ve druhém případě nebudou na výstupu žádná jména, protože **Select-Object** nemá na vstupu žádný objekt.

Sort-Object

Často se stane, že potřebujete zjistit např. největší soubory v adresáři nebo procesy, které nejvíce zatěžují počítač. V těchto případech se hodí cmdlet **Sort-Object**. Některé příklady jsem uváděl v předchozím díle a proto je nyní pouze krátce připomenou:

```
PS C:\> Get-Process | Sort-Object WorkingSet -Descending | Select-Object Name, WorkingSet -First 5
```

Zde jsme seřadili procesy podle využití fyzické paměti (sestupně) a pět největších žroutů jsme vypsalí.

Measure-Object

Potřebujete statistiku vašich skriptů? **Measure-Object** je vhodným kandidátem:

```
PS C:\> dir *.ps1 | Measure-Object Length -Maximum -Minimum -Average -Sum

Count      : 48
Average    : 3188,04166666667
Sum        : 153026
Maximum    : 27221
Minimum    : 8
Property   : Length
```

Group-Object

Tento cmdlet se vám bude hodit, pokud chcete například zjistit počet souborů v adresáři podle typu:

```
PS C:\> dir *.ps1 | Measure-Object Length -Maximum -Minimum -Average -Sum

Count      : 48
Average    : 3188,04166666667
Sum        : 153026
Maximum    : 27221
Minimum    : 8
Property   : Length
```

Compare-Object

Často se stane, že potřebujete porovnat určitý stav v čase. Nedávno jsem například řešil [porovnání členství ve skupinách v AD](#). Jednoduše si porovnání můžete vyzkoušet následujícím způsobem:

```
PS C:\> $a = Get-Process
PS C:\> notepad.exe
PS C:\> $b = Get-Process
PS C:\> Compare-Object -ReferenceObject $a -DifferenceObject $b

InputObject      SideIndicator
-----
System.Diagnostics.Process (notepad) =>
```

Zde jsme si do proměnné **\$a** uložili počáteční stav běžících procesů. Poté jsme spustili Notepad a do proměnné **\$b** jsme uložili aktuální stav. Pomocí **Compare-Object** jsme oba stavy porovnali a ve výpisu je vidět rozdíl. Takovýmto jednoduchým způsobem můžete například dělat audit na vašem serveru.

Hezký skript, jehož středem je **Compare-Object**, je například k nalezení v článku [Compare AD against snapshots](#) od [Dmitryho Sotnikova](#) (PowerShell MVP).

New-Object, Tee-Object

Tyto dva cmdlety asi nebudete ze začátku moc používat, ale jelikož patří do skupiny s ostatními alespoň se o nich zmíním.

New-Object slouží k vytváření vlastních objektů. Pokud máte například organizační strukturu, se kterou chcete pracovat v PowerShellu, není nic jednoduššího než si pro každého člověka vytvořit objekt a poté s nimi pracovat v rouře.

```
PS C:\> $companyStructure | where {$_.FirstName -eq 'David'} | Measure-Command
Count      : 9
```

Zde proměnná **\$companyStructure** obsahuje seznam lidí ve firmě (vytvořený pomocí **New-Object**). Dále je již práce stejná jako se všemi dalšími objekty.

Tee-Object se používá uprostřed roury, pokud chcete „uschovat“ aktuální stav objektů a zároveň je poslat do další roury.

```
PS C:\> Get-Process | Sort Id | Tee c:\Temp\psid.txt | Select -First 5
```

Na výstupu je pouze pět objektů, ale v souboru psid.txt jsou všechny objekty řazené podle Id.

ALIASES

V minulém díle jsem v posledním příkladu slíbil, že se dnes podíváme na aliasy. Alias můžeme chápat jako „zkratku“ při zápisu příkazů. Například

```
PS C:\> Get-WmiObject Win32_Process
```

můžeme pomocí aliasu přepsat jako

```
PS C:\> gwmi Win32_Process
```

Pokud jste v PowerShellu někdy zkusili příkaz **dir**, použili jste vlastně cmdlet **Get-ChildItem**. Seznam všech dostupných aliasů, můžete vypsát pomocí cmdletu **Get-Alias**.

```
PS C:\> Get-Alias
```

Pokud chcete použít opačnou cestu, tedy zjistit všechny aliasy pro daný cmdlet, můžete použít následující příkaz. Ukážeme si v něm aliasy pro cmdlety, které jsme probírali v předchozí kapitole.

```
PS C:\> Get-Alias | `
>> Where-Object {$_.Definition -like '*-Object'} | `
>> Format-Table Definition, Name -AutoSize
>>

Definition      Name
-----
Compare-Object  diff
ForEach-Object  foreach
ForEach-Object  %
Group-Object    group
Select-Object   select
Sort-Object     sort
Tee-Object      tee
Where-Object    where
Where-Object    }
```

Vypíšeme všechny aliasy a pomocí **Where-Object** vyfiltrujeme pouze ty, které splňují naše podmínky. Výsledky pak pomocí **Format-Table** přehledně vypíšeme. Předchozí příklad můžeme přepsat pomocí aliasů následovně:

```
PS C:\> gal | ?{$_Definition-like '*-ob*'} | ft n*,def* -a
```

WTH? Zde jsem se úmyslně vrhl do absolutních obskurností. Takový zápis byste asi nikde vidět neměli. Chtěl jsem ukázat nebezpečí aliasů (a dalších „zkracovačů“). Pokud to přeženete, těžko se ve výsledku vyznáte. Na aliasy existují rozporuplné názory – někdo je nepoužívá vůbec, někdo částečně a někdo se v nich vyžívá. Zde bych asi doporučil obecně uznávané pravidlo. Ve skriptech používejte celé jméno, v konzoli se aliasům nebraňte (ale samozřejmě platí – používejte to, co nejvíce vyhovuje vám).

DOLUJEME DATA ANEB JAK NA WMI

Než začneme s dnešní dávkou informací, chtěl bych se zmínit o změně, která se udála ve světě PowerShellu od vydání posledního dílu tohoto seriálu. 27. října byl uvolněn PowerShell v2 pro platformy Windows XP a Windows Server 2003 a nyní je tedy nová verze PowerShellu dostupná pro všechny hlavní platformy od Microsoftu. PowerShell v2 můžete (a vřele vám to doporučuji) stahovat z <http://support.microsoft.com/kb/968929> Vzhledem k této změně budu od tohoto dílu používat pro všechny příklady novou verzi. Zrovna u WMI se nám to bude velice hodit!

WMI (Windows Management Instrumentation) – technologie vynikající, leč – bohužel – částí administrátorů nenáviděná nebo (!) nepoznaná. Pro potřeby tohoto článku se spokojíme se zjednodušenou definicí, že WMI je technologie sloužící ke správě Windows systémů. Pokud byste se chtěli dozvědět víc, zkuste například mé oblíbené MSDN: [http://msdn.microsoft.com/en-us/library/aa394582\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394582(VS.85).aspx)

Pro potřeby tohoto článku budu předpokládat, že znáte základní informace o WMI a že víte, co je třída, vlastnost, WQL nebo instance.

1.1 Jak získat data z WMI

Náš hlavní kamarád pro dnešek bude gwmi – co to asi může být? Jedná se o alias - malý test, jestli si pamatujete minulý díl

```
PS C:\> Get-Alias gwmi
```

Get-WmiObject je – dle mého názoru – základním příkazem pro práci s WMI v PowerShellu. Ve v1 byl pouze tento cmdlet. Ve v2 jsou navíc tyto cmdlety: Invoke-WmiMethod, Remove-WmiObject, Register-WmiEvent, Set-WmiInstance.

Pojďme si ukázat základní použití cmdletu Get-WmiObject. Potřebujeme například zjistit typ našeho počítače:

```
PS C:\> Get-WmiObject Win32_ComputerSystem | Select-Object Model

Model
-----
HP Mini 5101
```

Jednoduché, efektivní. Pokud jste někdy pracovali s WMI ve VB Scriptu, musí vám předchozí příklad připadat jako zázrak. Ve VBS byla práce s WMI – řekněme – trochu komplikovaná (já osobně jsem do WMI přistupoval s lehkým odporem). S PowerShellem se situace změnila. Get-WmiObject je prostě dar z nebes (možná trochu přeháním, ale vzhledem k tomu, že s WMI pracuji každý den, vím, o čem mluvím).

Při práci s WMI vás budou asi nejvíce zajímat třídy, jejichž jméno začíná na Win32. Můžeme si je vypsat tímto příkazem:

```
PS C:\> Get-WmiObject -List | Where-Object {$_.Name -like 'Win32_*'} | Format-Wide -Column 3
```

Nebudu zde vypisovat všechny vrácené třídy, zkuste si je vypsat sami. Na mém počítači jich je 476. Zajímavé jsou například (kromě již zmíněné Win32_ComputerSystem) i Win32_OperatingSystem, Win32_Product nebo Win32_QuickFixEngineering (+ dalších asi 20, které budete používat nejčastěji, ale zkuste je najít sami – každému bude vyhovovat něco jiného dle zaměření). Pojdme si ukázat ještě jeden krátký příklad a poté se podíváme na složitější konstrukce. Bude nás zajímat verze operačního systému a service pack.

```
PS C:\> Get-WmiObject Win32_OperatingSystem | Format-Table Caption, CSDVersion
```

Caption	CSDVersion
-----	-----
Microsoft Windows XP Professional	Service Pack 3

1.2 Parametry cmdletu Get-WmiObject

Jeden z prvních parametrů, který určitě využijete je -computername. Již v PowerShellu v1 bylo možné je připojit na vzdálený počítač a zjistit informace z WMI. Stačilo použít právě tento parametr. Pokud si například chcete udělat inventuru instalovaných service packů na vaší síti, stačí použít následující příklad:

```
PS C:\> gwmi Win32_OperatingSystem -ComputerName pc1, pc2, pc3 | Format-Table Caption, CSDVersion - AutoSize
```

Daleko efektivnější než psát jména počítačů do konzole bude zřejmě čtení z textového souboru, takže náš příklad upravíme následovně:

```
PS C:\> gwmi Win32_OperatingSystem -ComputerName (Get-Content ./computers.txt) | Format-Table Caption, CSDVersion -AutoSize
```

Při čtení některé z objemnějších tříd v WMI je lepší využít některý z následujících parametrů: **Filter** nebo **Query**. Důvody jsou dva – rychlost zpracování a objem přenášených dat (ve spojení s parametrem ComputerName). Ukážeme si tři různé přístupy k získání požadovaných informací a povíme si o jejich výhodách a nevýhodách. Úkolem bude zjistit, jestli máme na počítači nainstalován produkt PowerGUI (mimořádně vynikající, volně dostupný editor na PowerShell).

```
PS C:\> Get-WmiObject Win32_Product | Where-Object {$_.Name -like '*PowerGUI*'}
```

Pro zjištění instalovaných programů použijeme třídu Win32_Product. Prvním vaším nápadem bude možná útok „hrubou silou“ – gwmi Win32_Product – v záplavě ujíždějících obrazovek si poté zpětně najdete (nebo nenajdete) požadovanou informaci. Toto je samozřejmě cesta, ale pro naše potřeby ji budeme považovat za nevhodnou (i když někdy je tento styl to nejrychlejší, co v praxi máte).

Jako další by vás mohlo napadnout použít filtrování objektů z minulého dílu a použít Where-Object.

```
PS C:\> Get-WmiObject -Class Win32_Product -Filter 'Name LIKE "%PowerGUI%''
```

V tomto případě by se nám měl vrátit záznam pouze pro zmiňované PowerGUI. Tato metoda není špatná, ale má jednu vadu. PowerShell vnesle dotaz do WMI, dostane zpět **všechna** data z třídy **Win32_Product** a poté je pomocí cmdletu Where-Object filtruje na základě našeho požadavku. Problém máme právě s oním slůvkem „všechno“. Objekty opravdu filtrujeme až na úrovni PowerShellu, což není úplně efektivní. Lepší je, pokud filtrujeme data přímo na úrovni WMI (čímž zrychlujeme provádění dotazu). Zpátky se nám pak vrátí pouze data splňující zadanou podmínku. Což v případě přístupu na vzdálený počítač snižuje zatížení sítě. Použijeme tedy parametr Filter.

```
PS C:\> Get-WmiObject -Query 'SELECT * FROM Win32_Product WHERE Name LIKE "%PowerGUI%''
```

Nyní jsme přenesli filtrování dovnitř WMI a zpět se nám vrací **pouze** data splňující podmínku uvnitř filtru. Zde již potřebujeme alespoň základní znalosti WQL ([http://msdn.microsoft.com/en-us/library/aa394606\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394606(VS.85).aspx)). Pokud chceme mít úplnou kontrolu nad filtrováním, můžeme použít parametr Query a příklad přepsat následujícím způsobem.

```
PS C:\> Get-WmiObject -Query 'SELECT * FROM Win32_Product WHERE Name LIKE "%PowerGUI%''
```

Nyní jsme použili kompletní dotaz pomocí WQL. Uvědomte si, že vše, co se nám vrací je opět objekt a proto můžeme použít například následující konstrukci:

```
PS C:\> $verze = (Get-WmiObject -Query 'SELECT * FROM Win32_Product WHERE Name LIKE "%PowerGUI%') .Version
PS C:\> if ($verze) {Write-Host "Nainstalovana verze: $verze"}

Nainstalovana verze: 1.9.5.966
```

Pokud se chcete podívat na srovnání rychlosti zmiňovaných přístupů, podívejte se například na <http://powershell-cz.blogspot.com/2009/08/ziskavani-informaci-z-wmi.html>

Osobně používám Get-WmiObject velmi často, a většinou používám buď první (hrubá síla) nebo poslední (Query) přístup. Při použití obou přístupů většinou ukládám vrácené objekty do proměnné, se kterou dále pracuji.

1.3 Parametr *AsJob*

V PowerShellu v2 se objevila (mimojiné) možnost spouštět cmdlety na pozadí. Toto je obrovská výhoda při spouštění příkazů, které trvají dlouhou dobu. Můžete si běh příkazu přepnout do pozadí a dále pracovat v konzoli bez jejího blokování běžícím příkazem. Pokud budeme chtít poslední příkaz vylepšit, použijeme tedy

```
PS C:\> Get-WmiObject -Query 'SELECT * FROM Win32_Product WHERE Name LIKE "%PowerGUI%'' -AsJob
```

Id	Name	State	HasMoreData	Location	Command
1	Job1	Running	False	localhost	Get-WMIObject

Dotaz do WMI se přesune na pozadí a okamžitě se objeví příkazový řádek. Můžete pracovat dále a za chvíli zjistit stav běžícího dotazu pomocí cmdletu Get-Job.

```
PS C:\> Get-Job
```

Id	Name	State	HasMoreData	Location	Command
1	Job1	Running	False	localhost	Get-WMIObject

Je vidět, že job nám ještě stále běží. Pokud skončí, uvidíte následující výsledek:

```
PS C:\> Get-Job
```

Id	Name	State	HasMoreData	Location	Command
1	Job1	Completed	True	localhost	Get-WMIObject

State je nyní Completed a proto si můžeme zobrazit výsledek

```
PS C:\> Receive-Job -Id 1
```

IdentifyingNumber : {03172308-0A39-4084-BA66-2F954E192451}
Name : Quest PowerGUI 1.9.5
Vendor : Quest Software, Inc.
Version : 1.9.5.966
Caption : Quest PowerGUI 1.9.5

Receive-Job přináší jedno nebezpečí. V případě, že jej použijete, jak bylo naznačeno, data obdržíte zpět, ale pokud si je neuložíte, znovu už je nezískáte. Zkuste zavolat příkaz znovu a uvidíte. Máte dvě možnosti: 1) uložit si výstup cmdletu Receive-Job do proměnné nebo 2) zavolat Receive-Job s parametrem -Keep.

Závěrem jedna malá odbočka k testovanému PowerGUI. Pokud se chcete podívat, jak přistupovat do WMI přes PowerGUI, stáhněte si tento produkt z www.powergui.org a prozkoumejte větev WMI

Browser. Vynikajícím zdrojem pro průzkum WMI z PowerShellu je také WMI Explorer on [\a](#).

Dnes jsme se lehce podívali na jednu novinku v PowerShellu v2 – joby. Vzhledem k tomu, že verze 2 je nyní standard budu ji v dalších dílech využívat více. Po plánovaném posledním díle se zaměříme na novinky, které nám – administrátorům – výrazně ulehčují nebo zrychlují práci.

SOUBOROVÝ SYSTÉM A REGISTRY

Po minulém díle, kdy jsme se podívali do hloubi systému si dnes dáme trochu oddechovější část – práci se souborovým systémem a registrem. Nejdříve se ale podíváme na ideu takzvaných PSDrives.

„Vše je drive.“ By mohla znít parafráze známého „vše je soubor“.

Již ve verzi 1 PowerShellu byly PSDrives zavedeny. Jedná se o snahu jak zjednodušit přístup ke stejným/podobným částem systému tak, abyste je již jednou naučenou věc nemuseli učit znovu. Jako příklad porovnáme souborový systém a registr. Vžijte se do doby středověku Windows (tedy doby, kdy ještě neexistoval PowerShell). Pokud pomineme Windows Explorer a Registry Editor (ale jako administrátoři jste je stejně nepoužívali), mohli jste použít cmd.exe a reg.exe. Oba dva příkazy určitě znáte, ale dovoluji si tvrdit, že reg.exe používáte velmi málo už jen z toho důvodu, že jeho syntaxe je „prostě jiná“ (netvrdím, že REG QUERY používám každý den 100x – regedit vede na celé čáře). PSDrives přináší výhodu v tom, že se zbavujeme onoho „prostě jiná“ a pohybujeme se v registru (ale nejen tam, jak uvidíme vzápětí) pomocí příkazů známých z cmd.exe. Čili převedeno do řeči PowerShellu, jestliže vím, že příkazem

```
PS C:\> dir c:\temp
```

Vylistuji obsah adresáře, můžu příkazem

```
PS C:\> dir HKLM:\SYSTEM
```

Vylistovat obsah registru. A co teprve zábava s příkazem

```
PS C:\> dir cert:\LocalMachine\AuthRoot
```

Který zobrazí certifikáty. Chcete-li zjistit, co vše je vlastně PSDrive, můžete zkusit cmdlet **Get-PSDrive** (případně tématickou nápovědu **about_Providers**). Měli byste vidět minimálně následující seznam:

```
PS C:\> Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root
Alias			Alias	
C	33.28	199.59	FileSystem	C:\
cert			Certificate	\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
Variable			Variable	
WSMan			WSMan	

Pokud používáte PowerShell v1, nebudete mít v seznamu poslední položku (WsMan). Nyní vidíte, proč jsem mohl použít **dir hklm:** a **dir cert:** - stejným způsobem můžete prozkoumat například aliasy nebo funkce. Určitě se vám bude hodit náhrada za příkaz set, zkuste

```
PS C:\> dir env:

Name                           Value
----                           -
ALLUSERSPROFILE                C:\Documents and Settings\All Users
CLIENTNAME                     Console
CommonProgramFiles             C:\Program Files\Common Files
COMPUTERNAME                    NETBOOK
ComSpec                         C:\WINDOWS\system32\cmd.exe
... (zkráceno)
windir                         C:\WINDOWS
```

Základy souborového systému

Ukážeme si nyní základní příkazy pro pohyb a práci v souborovém systému (a tedy i ostatních PSDrives). Souborový systém volím z toho důvodu, že je asi nejnázornější pro výklad.

Základem jsou příkazy známé již od dob MS-DOSu (dir, cd, copy, ...) nebo například z lixových OS (ls, man, ...). V PowerShellu jsou tyto příkazy definovány jako aliasy, např. **dir** je alias pro **Get-ChildItem**. Vyzkoušejte si příkazy, které znáte z cmd.exe, a zkuste vypátrat jejich cmdlety. Můžete samozřejmě hledat i cmdlety pro příkazy copy, move, rm, ... Brzy zjistíte, že končí slovem item, čili můžete vyzkoušet

```
PS C:\> Get-Command *-item

CommandType      Name                Definition
-----
Cmdlet            Clear-Item           Clear-Item [-Path] <String[]> [-Force] [-Filter ..
Cmdlet            Copy-Item            Copy-Item [-Path] <String[]> [[-Destination] <St..
Cmdlet            Get-Item             Get-Item [-Path] <String[]> [-Filter <String>] ..
Cmdlet            Invoke-Item          Invoke-Item [-Path] <String[]> [-Filter <String>..
Cmdlet            Move-Item            Move-Item [-Path] <String[]> [[-Destination] <St..
Cmdlet            New-Item             New-Item [-Path] <String[]> [-ItemType <String>]..
Cmdlet            Remove-Item          Remove-Item [-Path] <String[]> [-Filter <String>..
Cmdlet            Rename-Item          Rename-Item [-Path] <String> [-NewName] <String>..
Cmdlet            Set-Item             Set-Item [-Path] <String[]> [[-Value] <Object>] ..
```

U většiny z nich není asi potřeba něco dodávat. Trošku výjimečným případem je **Invoke-Item**, který spustí defaultní akci nad volaným objektem (např. otevře textový soubor v editoru), čili

```
PS C:\> New-Item -Path soubor.txt -ItemType file -Value "PowerShell je cool!" | Invoke-Item
```

vytvoří textový soubor, naplní jej zadaným textem a poté jej otevře v editoru, který je nastaven jako primární pro otevírání textových souborů. Na mém počítači je výstupem toto okno:



1.3 Práce s ACL

Při mých prezentacích pro administrátory sklízí asi největší obdiv příkazy pro práci s ACL. Vždy je ukazují na následujícím příkladu. Máme za úkol vytvořit adresářovou strukturu pro nově vzniklá oddělení. Potřebujeme adresáře od písmena A do Z a pro tyto adresáře musíme nastavit předem daná práva (různá dle oddělení). V Exploréru noční můra, v PowerShellu zábava. Všimněte si i použití ostatních cmdletů určených pro práci s PSDrives.

```
# Vytvoření nového adresáře
New-Item -Path 'c:\temp\PSTestRoot' -Type directory

# Vytvoření nového PSDrive typu souborový systém a přechod do takto vytvořené struktury
New-PSDrive -Root C:\temp\PSTestRoot -Name PSTestRoot -PSProvider FileSystem
Push-Location PSTestRoot:

# Vytvoření adresářů 'a' až 'z'
97..122 | % {New-Item -name ([char]$_) -Type Directory}

# Vytvoření podadresářů v každém z vytvořených adresářů
ls |% {New-Item -Path (Join-Path $_ "IS") -Type directory}
ls |% {New-Item -Path (Join-Path $_ "TAX") -Type directory}

# Dle již vytvořené šablony adresáře se všemy nastavenými právy si vytvoříme objekty,
# které ponese tato práva

# Proměnné $aclIS a $aclTAX jsou třídy System.Security.AccessControl.DirectorySecurity
$aclIS = Get-Acl 'c:\Temp\PSTestRootTemplate\TemplateIS'
$aclTAX = Get-Acl 'c:\Temp\PSTestRootTemplate\TemplateTAX'

# Můžeme si zobrazit nastavená práva
$aclIS | fl ac*g

# Nyní uložená práva nastavíme na cílových adresářích
ls -r |? {$_.FullName -match 'IS$'} | Set-Acl -AclObject $aclIS
ls -r |? {$_.FullName -match 'TAX$'} | Set-Acl -AclObject $aclTAX

# Přepneme zpět do původního adresáře
Pop-Location

# Provedeme závěrečný úklid
Remove-PSDrive -Name PSTestRoot
Remove-Item -Path 'c:\temp\PSTestRoot' -Recurse -Force -WhatIf
Remove-Item -Path 'c:\temp\PSTestRoot' -Recurse -Force
```

Chcete-li více informací k tomuto příkladu i celé prezentaci, můžete se podívat na [tento příspěvek](#).

Pokud spravujete file server, určitě se vám budou cmdlety **Get-Acl** a **Set-Acl** hodit. Vzhledem k tomu, že je můžete využívat v rouře stanou se vám určitě výtannými pomocníky. Nezapomeňte na PSDrives – oba dva příkazy můžete použít také pro práci s registry.

Registr

Jelikož cmdlety pro práci s registry jsou stejné jako pro práci se souborovým systémem, podíváme se rovnou na jedno úskalí. Při práci s registry nás čeká jedno nepříjemné překvapení. Zkusme:

```
PS C:\> cd HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run> dir

Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

SKC  VC Name                                Property
---  --
3    1 OptionalComponents                  {(default)}
```

Místo očekávaného seznamu programů spouštěných při startu systému dostáváme jakýsi zvláštní, nicneříkající výpis. Pokud chceme zjistit požadované informace, musíme použít jiný cmdlet – hodnoty

v určitém klíči jsou totiž považovány za vlastnosti tohoto klíče (v našem případě větev Run). Použijeme proto **Get-ItemProperty** a výsledkem bude očekávaný výstup

```
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\run> Get-ItemProperty .

PSPath          : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Current
                  Version\run
PSParentPath     : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Current
                  Version
PSChildName      : run
PSDrive          : HKLM
PSProvider       : Microsoft.PowerShell.Core\Registry
MsmqIntCert      : regsvr32 /s mqrt.dll
IAAnotif         : C:\Program Files\Intel\Intel Matrix Storage Manager\iaanotif.exe
IgfxTray         : C:\WINDOWS\system32\igfxtray.exe
HotKeysCmds      : C:\WINDOWS\system32\hkcnd.exe
Persistence      : C:\WINDOWS\system32\igfxpers.exe
SynTPEnh         : C:\Program Files\Synaptics\SynTP\SynTPEnh.exe
MSSE             : "c:\Program Files\Microsoft Security Essentials\msseces.exe" -hide
```

Jelikož pracujeme opět s objekty, není problém získat konkrétní hodnotu nějakého klíče

```
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\run> (Get-ItemProperty .).MSSE
"c:\Program Files\Microsoft Security Essentials\msseces.exe" -hide
```

Další PSDrives

Jak by se vám líbilo, kdybyste mohli požit následující příkaz:

```
PS C:\> cd ad:
```

Zde parazituji na názvu jiného článku (<http://www.jonathanmedd.net/2009/11/cd-ad-wow.html>), ale asi tušíte, co vás čeká. Přístup do Active Directory přes PSDrives a práce s účty podobná jako se soubory. Pokud vás práce v AD před PowerShell zajímá více, měli byste věnovat svou pozornost [tomuto blogu](#). Pokud ještě nemáte doménu s Windows 2008 R2 serverem nebo nepoužíváte [Active Directory Management Gateway Service](#), můžete použít [PSCX](#), které PSDrive pro Active Directory také obsahuje.

Pro administrátory SQL Serveru 2008 není například neznámý PSDrive SQLSERVER.



PS PRO PROGRAMÁTORY

Na Technet Flash běží už několikátý díl o skriptovacím nástroji zvaném PowerShell, kterým provází [David Moravec](#). Seznámili jsme se se základy PowerShellu a dotknuli se i pokročilejších témat jako jsou WMI anebo PsDrives.

PowerShell byl navrhnutý primárně pro administrátory, kteří dostali do rukou konečně rozumný nástroj umožňující zautomatizovat jim jejich postupy. Proto jej Microsoft integruje například do Windows Serveru 2008, nebo do SQL Serveru 2008. Vznikají i další knihovny – na správu IIS, Exchange, VmWare, atd.

Proč by ale PowerShell neměl dostat šanci i u programátorů, nebo pokročilých uživatelů Windows? Pokud jste někdy tvořili .bat soubor, nebo se snažili o „skriptování“ v klasické windows command line, bude PowerShell určitě správnou volbou. Jeho syntaxe je daleko

příjemnější, více příbuzná běžným programovacím jazykům. Krom toho disponuje i velmi zajímavými příkazy (*cmdlety*) přímo integrovanými do jádra.

Tento článek má být určen především programátorům. Budu se zde snažit nastínit některé tipy, které jim mohou usnadnit práci.

Automatické zpracování

Jako první asi zdůrazním tu základní věc, proč PowerShell získal takovou oblibu – velmi jednoduše můžeme vyřešit úlohu, kterou bychom museli jinak psát například v C#. Co znamená "jednoduše"? Jediné, co musíte udělat, je vytvořit si *ps1* soubor a tento spustit v PowerShell konzoli. (samozřejmě bychom se obešli i bez souboru, ale takto si postupně můžeme tvořit knihovnu užitečných skriptů)

Jak bychom to dělali jinak? Ve Visual Studiu bychom vytvořili C# command line projekt a do příslušné vygenerované *Main* metody bychom začali psát kód. Parsování předaných argumentů, procházení souborů, adresářů, operace nad nimi, práce s chybovými stavy atd. Přeložit, spustit a máme výsledný *exe* soubor. Na jednoduchou úlohu je toto příliš zdlouhavé a množství vygenerovaných souborů je velké (i případě, že bychom pouze překládali pomocí *csc*, počet souborů je dvojnásobný).

C# se hodí na komplexní zpracování, kde využijeme jeho aspektů. Už jen *code noise* je u C# ve srovnání s PowerShellem obrovský. Jakýkoliv úkol proveditelný z .NETu, se dá zpracovat pomocí PowerShellu. Typickým úkolem může být: projdi všechny soubory v daném adresáři, otevři je a nahraď řetězec XYZ řetězcem ABC.

Samozřejmě se nemusíme omezovat jen na nahrazování. Nedávno jsem viděl člověka, který řešil duplicitu u svých fotek. Na disku měl velmi mnoho fotografií a některé byly v různých složkách vícekrát. Jeho přáním pak bylo najít ty fotografie, které jsou stejné. Přitom bylo potřeba u nich ignorovat i Exif – některé totiž byly s Exifem, jiné bez. Můžete se podívat na [daný dotaz](#).

Jinak pro jednu velmi jednoduchou ukázkou si můžeme roztřídit fotografie podle toho, jestli jsou navysoko, nebo naležato. Výsledek si pak seřadíme podle orientace a jména. Fotografie mohou být v adresáři libovolně hluboko zanořené:

```
$d = gci D:\temp\ dscn*.jpg -rec |

% {

    $b = new-object System.Drawing.Bitmap $_.FullName

    New-Object PsObject -prop @{

        Orientation=if($b.Height -gt $b.Width) { 'v' } else {'h' };

        Path=$_.fullname }

    $b.Dispose()

} |

sort -Property Orientation,Path
```

Kód by se dal ještě zkrátit nevytvářením objektu, ale už by to bylo případně na úkor čitelnosti. Podobných úloh pak můžeme najít mnoho, stačí zapojit svou fantazii. Dále už budu mluvit o konkrétních technikách a vlastnostech PowerShellu.

Práce s XML

Jedna z věcí, která se vám zalíbí, je práce s XML. Představte si, že se na celý XML soubor podíváme jako na objekt – z atributů se stanou stringové property a z vnořených elementů se stanou složené property objektu. Leckoho asi napadne (*de*)*serializace*. V tomto případě sice o *deserializaci* nejde, ale výsledek je podobný. Při *deserializaci* je vytvořen objekt určitého typu, zatímco při načtení xml v PowerShellu dostaneme k dispozici objekt typu `XmlDocument`, který je všem programátorům v .NET dobře známý.

Vezměme si příklad. Nejdřív si vytvoříme testovací xml soubor.

```
PS> @"
<root>

  <article date="2010-12-01">

    <name>Discover new dimensions</name>

    <body>Discover them now. Go!</body>

  </article>

  <article date="2000-01-01">

    <name>Future</name>

    <body>what will be with us in ten years?</body>

  </article>

</root>

"@ | Set-Content c:\temp\flashtest.xml
```

Zkusíme soubor načíst a přetypovat na xml (jde o použití tzv. *acelerátoru*) a podívat se, co se nám vlastně vrátilo.

```
PS>$x = [xml](gc c:\temp\flashtest.xml)

PS>$node = $x.root.article | ? { [datetime]$_ .date -lt [datetime]'2005-01-01' }

PS>$node

date          name          body
----          -
2000-01-01    Future        what will be with us in ten years?
```

```
PS>$node.name = 'Near ' + $node.name

PS>$x.root.article[0].date = (get-date).ToString('yyyy-MM-dd')
```

Pomocí tečkové notace známé snad všem programátorům přistupujeme k jednotlivým elementům. Proměnná `$x` je typu `XmlDocument`, zatímco `$node` je typu `XmlElement`. Nastavení nové hodnoty atributu, nebo testového elementu probíhá jednoduchým přiřazením. Přidávání elementů je již plně v režii .NET.

```
PS>$note = $x.CreateElement('note')

PS>$note.InnerText = 'poor content'

PS>$node.AppendChild($note)

PS>$x.root.article[1]
```

date	name	body	note
----	----	----	----
2000-01-01	Near Future	what will be with us in ten years?	poor content

Vybírat elementy pomocí *XPath* šlo dříve opět jen pomocí .NETích prostředků, od verze 2 máme k dispozici cmdlet `Select-Xml`.

```
PS>Select-Xml -Xml $x -XPath '//article[contains(body/text(),"ten")]'

PS>#použití namespace

PS>$ns = @{ e = "http://www.w3.org/1999/xhtml" }

PS>Select-Xml -Path $somePath -XPath //e:div[@id] -names $ns
```

Jako zdroj pro `Select-Xml` můžeme použít nejen xml, ale i seznam cest k xml souborům, nebo xml ve formě řetězce (`string`).

Na uložení xml nemá PowerShell žádné speciální prostředky. Použijeme opět .NET volání:

```
PS>$x.Save('c:\temp\res.xml')
```

Regex tester

V PowerShellu si můžete velmi rychle vyzkoušet své regulární výrazy. Nebudu zde popisovat, co je to regulární výraz, jak se tvoří a k čemu slouží. Více informací k regulárním výrazům nabízí například Regular-expressions.info.

Místo vysvětlování základů se podíváme, jak můžeme rychle otestovat, zda regulární výraz "dělá to, co má".

Jestliže máte po ruce konzoli PowerShellu, je toto snad nejrychlejší způsob (pozn. pokud se vám konzole špatně hledá mezi ostatními okny, možná vám pomůže [AutoHotkey](#)). Spouštět programy určené pro testování regulárních výrazů nebo si je zkoušet online vyžaduje režii, která programátora zpomalí. Jak uvidíme, PowerShell touto brzdou nebude. Podívejme se tedy, jak na to – pro testování regulárních výrazů je zde totiž několik přístupů.

Operátor -match

Nejjednodušším z nich je použití operátoru `-match`, kterému jako pravý operand předáme regulární výraz.

```
# hloupý regex jen pro účel demonstrace

PS>'jeho email je karel@novak.cz' -match ' \w+@[a-zA-Z_]+\.(?<d>[a-zA-Z]{2,3}) '

True

PS>$matches

Name                                Value
----                                -
d                                    cz
0                                    karel@novak.cz
```

V kolekci `$matches` nám PowerShell udržuje groupy z posledního vyhodnocení operátorem `-match`. Groupa 0 obsahuje celý řetězec, který regulárnímu výrazu odpovídá. Regulární výraz není case sensitive a v podstatě odpovídá regulárnímu výrazu bez jakýchkoliv speciálních options.

Pro případ, že potřebujeme mít test case-sensitive, použijeme operátor `-cmatch`.

Operátor `-match` také umí filtrovat pole řetězců. Vyzkoušejte si následující příklad a hned pochopíte:

```
PS>'1a','2b','1c' -match '1\w'

PS>'1a' -match '1\w'
```

Akcelerátor [regex]

Na vytvoření regulárního výrazu můžete použít akcelerátor `[regex]`:

```
PS>$r = [regex] '^ \w+@[a-zA-Z_]+\.(?<d>[a-zA-Z]{2,3}) $'

PS>$r.GetType().FullName

System.Text.RegularExpressions.Regex
```

Vytvoří se klasický .NET regulární výraz a uloží do proměnné `$r`. Opět zde nemá žádné zvláštní options, o čemž se můžeme přesvědčit takto:

```
PS>$r | fl
```

S regulárním výrazem pak pracujeme tak, jak známe – pokud bychom nevěděli, pak nám pomůže výpis metod a properties:

```
PS>$r | gm
```

```
TypeName: System.Text.RegularExpressions.Regex

Name                MemberType Definition
-----
...
GetGroupNames       Method      string[] GetGroupNames()
GetGroupNumbers     Method      int[] GetGroupNumbers()
...
IsMatch             Method      bool IsMatch(string input), bool
IsMatch(strin
Match               Method      System.Text.RegularExpressions.Match
Match(st
Matches            Method      System.Text.RegularExpressions.MatchCollection
Replace            Method      string Replace(string input, string
replacemen
...
```

Vytvoření objektu

Poslední možností, která ale úzce souvisí s druhou, je vytvoření regulárního výrazu pomocí cmdletu `new-object`. Zde pak můžeme specifikovat volby jako `multiline`, `singleline` apod.

```
PS>$opts = [System.Text.RegularExpressions.RegexOptions] 'MultiLine,
SingleLine'
PS>$r = new-object Text.RegularExpressions '^\w+@[a-zA-Z_]+?\.(?<d>
[a-zA-Z]{2,3})$', $opts
```

Povšimněte si zajímavé syntaxe, jak specifikovat options u regulárního výrazu – nenašel jsem ji pořádně v nápovědě popsanou. Pomohla mi pouze zmínka na blogu [Using Enumerated types \(Enums\) in PowerShell](#).

Obecně se s *enumy* v PowerShellu pracuje hezky. Není nutné uvádět typ enumu, stačí pouze hodnota ve stringové podobě.

Navíc – zkuste si například toto:

```
PS>[string]::Compare('a','a',[Globalization.CultureInfo]::CurrentCulture,
'test')

Cannot convert argument "3", with value: "test"..... The possible
enumeration values are "None, IgnoreCase, IgnoreNonSpace, ... Ordinal".

PS>[string]::Compare('a','a',[Globalization.CultureInfo]::CurrentCulture,
'IgnoreCase')
```

PowerShell vám napoví, jaké hodnoty může enum nabývat – není tedy nutné hledat v dokumentaci. Výrazně se tím zrychlí vývoj.

Pozn.: Samozřejmě můžeme použít i statické metody třídy `[regex]`, které se volají pomocí dvojtečky:

```
PS>[regex]::IsMatch('karel@novak.cz', '^\\w+@[a-zA-Z_]+\\. (?<d>[a-zA-Z]{2,3})$')
```

Pozn. 2: Krom operátoru `-match` je vhodné znát i operátor `-replace`, který slouží k nahrazování textu. Velmi jednoduché a účinné nahrazení textu v souboru můžeme dosáhnout takto:

```
PS>(Get-Content c:\\test.txt) -replace 'abc','def' | Set-Content c:\\test.txt
```

Metoda *FindR*

Později v sekci o clipboardu využívám metodu `FindR`. Jde o funkci, která filtruje vstupní data podle zadaného regulárního výrazu.

```
filter FindR($regex) {
    [Regex]::Matches($_, $regex, 'IgnoreCase') | % { $_.Value }
}
```

Jako data můžeme poslat prakticky cokoliv. PowerShell se poté snažit vstup konvertovat na string, protože metoda `Matches` očekává string. Pokud do pipeline posíláme objekty, PowerShell runtime se rozhodne sám, jak objekt zkonvertuje na string.

Tak například `dir | findr '.*gs.*'` nám bude fungovat, ale bude vracet pouze jména souborů/adresářů.

Ale `Get-WinEvent -LogName Application -MaxEvents 10 | findr '.*instal.*'` nezafunguje, protože jako vstup do metody `Matches` PowerShell vloží řetězec `'System.Diagnostics.Eventing.Reader.EventLogRecord'`. Tuto hodnotu zřejmě získal prostým zavoláním `ToString()` na objektech z `Get-WinEvent`.

Reálné použití si sami jistě dokážete vymyslet. Já jej sem tam používám na práci s logy od zákazníků. Naposledy jsem filtr použil, když jsem zkoumal log, který se týkal importu souborů. V logu byly zapsány jejich velikosti a já jsem potřeboval zjistit jejich průměrnou velikost a celkový součet. Na vyparsování konkrétních velikostí je použit [look behind](#).

```
gc c:\dev\WO\wpdataimport.log |  
    findr -r '(?<=Job content file size: )\d+' |  
Measure-Object -Sum -Average
```

První příkaz načte soubor, druhý profiluje jeho řádky a vrátí řetězce (číslíce), před kterými je "Job content file size: " a poslední příkaz sečte čísla (zkonvertuje řetězce na čísla) a spočítá jejich průměr.

Kódování, dekódování, konverze, ...

Obzvláště weboví vývojáři někdy potřebují pracovat s *base64* stringy, případně kódovat a dekódovat url. Samozřejmě na konverzi existují dostupné nástroje. Jednodušší ale je přímo do PowerShell profilu přidat příslušné funkce, takže budou vždy velmi rychle dostupné.

```
# načte assembly potřebnou pro práci s url (pokud ještě načtená nebyla)  
Add-Type -AssemblyName System.Web  
  
function FromBase64([string]$str) {  
  
    [text.encoding]::utf8.getstring([convert]::FromBase64String($str))  
  
}  
  
function ToBase64([string]$str) {  
  
    [convert]::ToBase64String([text.encoding]::utf8.getBytes($str))  
  
}  
  
function UrlDecode([string]$url) {  
  
    [Web.HttpUtility]::UrlDecode($url)  
  
}
```

```
function UrlEncode([string]$url) {

    [Web.HttpUtility]::UrlEncode($url)

}

function HtmlDecode([string]$url) {

    [Web.HttpUtility]::HtmlDecode($url)

}

function HtmlEncode([string]$url) {

    [Web.HttpUtility]::HtmlEncode($url)

}
```

Ti kteří často pracují v PowerShellu pak mohou ocenit například takovou funkci:

```
PS> function Run-GoogleQuery {

    param([string]$word)

    Start-Process ('http://www.google.cz/search?q=' + (UrlEncode $word))

}

PS> Set-Alias qg Run-GoogleQuery

PS> qg 'toto je test' # spustí defaultní browser s dotazem na google
```

Konverzi `HtmlEncode` jsem například využil při psaní tohoto článku, když jsem potřeboval vložit PowerShell kód a zakódovat korektně HTML znaky. S využitím funkce `clip` (kterou uvedu později v sekci o clipboardu) to bylo vcelku jednoduché:

```
HtmlEncode (clip) | clip
```

(znamená to: vezmi obsah schránky, zakóduj a vlož do schránky)

Převod Html2Xml

Někdy se hodí i převod html na xml. V tomto případě využijeme volně dostupnou assembly *SgmlReader*.

```

function Convert-Html2Xml {
    param(
        [Parameter(ValueFromPipeline=$true)][object[]]$html
    )
    begin { $sb = new-object Text.StringBuilder(20kb) }
    process { $html | % { $null = $sb.AppendLine($_) } }
    end {
        # udelame si zivot jednodussi...

        $str = $sb.ToString().Replace(' xmlns="http://www.w3.org/1999/xhtml"',
'',
        Add-Type -Path G:\bin\SgmlReaderDll.dll

        $sr = new-object io.stringreader $str

        $sgml = new-object Sgml.SgmlReader

        $sgml.DocType = 'HTML';

        $sgml.WhitespaceHandling = 'All';

        $sgml.CaseFolding = 'ToLower';

        $sgml.InputStream = $sr;

        $xml = new-object Xml.XmlDocument;

        $xml.PreserveWhitespace = $true;

        $xml.XmlResolver = $null;

        $xml.Load($sgml);

        $sgml.Close()
    }
}

```

```

        $sr.Close()

        $xml

    }

}

```

Máme pak dva způsoby, jak volat:

```

PS>$x1 = gc c:\temp\testhtmlsource.delete.html | Convert-Html2Xml

PS>$x1.Save('c:\temp\test1.xml')

PS>$x3 = Convert-Html2Xml (gc c:\temp\testhtmlsource.html)

PS>$x3.Save('c:\temp\test2.xml')

```

Tuto funkci můžeme například použít, pokud daná webová služba nemá přístupné API, ale uživatel s ní pracuje pouze přes její webové UI. Příkladem může být <http://www.slovník.cz>. Kompletní [řešení](#) ve stylu quick&dirty zahrnuje i použití *XPath* pomocí cmdletu `Select-Xml`. Jiné řešení jsem rychle stvořil pro kolegu, který vytvářel seznam filům s linky na [CSFD](#).

nehledejte krásu, ale [automatizaci](#)

A ještě jeden rychlý příklad: narazili jste na blog, který má velmi mnoho příspěvků a do stránky se vypisují celé příspěvky. Může vypadat například [takto](#). Rádi byste věděli názvy jednotlivých článků, protože vás zajímá třeba o čem ten člověk píše? Samozřejmě můžete scrollovat, ale protože je stránka velmi dlouhá, hrozí nebezpečí, že něco zajímavého přeskočíte. Jaké je řešení?

```

$xml = Convert-Html2Xml (download-page 'http://www.nivot.org/CategoryView,
category,PowerShell.aspx' )

Select-Xml -Xml $xml -XPath '//div[@class="itemTitle"]/a/text()' | %
{ $_.Node.Value }

PowerShell 2.0 - About Dynamic Parameters

PowerShell 2.0 â€” Introducing the PModem File Transfer Protocol

PowerShell 2.0 - Enabling Remoting with Virtual XP Mode on Windows 7

PowerShell 2.0 goes RTM for ALL Platforms

```

PowerShell 2.0 - Module Initializers

PowerShell 2.0 â€" Getting and setting text to and from the clipboard

PowerShell 2.0 â€" Asynchronous Callbacks from .NET

PowerShell â€" Function Parameters & .NET Attributes

PowerShell 2.0: A Configurable and Flexible Script Logger Module

....

Clipboard

Na první pohled si možná řeknete: *"K čemu mi bude clipboard?"*. Velmi často jej používám jako transportní mechanismus z nějaké aplikace do PowerShellu. Už jste v článku několikrát viděli použití funkce `clip`, ale tedy ještě pro zopakování ještě poslední příklad:

Mám problém s deadlocky na sql serveru. Spustím si jej tedy z konzole s přepínači kvůli detekci deadlocků. Poté, co se mi podaří nasimulovat deadlock, označím si obsah konzole sqlserveru a jdu do PowerShellu. Chci například zjistit jednotlivá SPID. Jak na to?

```
(clip) | FindR -r 'SPID: \d+'
```

Toto mi vypíše ID všech zúčastněných procesů (pro vynechání duplicit bychom použili `select -unique`).

Vím, že v SQL analyzeru je možné SPID také vidět, berte to jen jako příklad.

Podobných případů jistě naleznete dost. Funkce na práci s clipboardem máme tedy tyto:

```
Add-Type -a system.windows.forms
```

```
function Set-Clipboard {
```

```
    param(
```

```
[Parameter(Mandatory=$true, ValueFromPipeline=$true, Position=0)][object]$s
    )
```

```
    begin { $sb = new-object Text.StringBuilder }
```

```
    process {
```

```
        $s | % {
```

```
            if ($sb.Length -gt 0) { $null = $sb.AppendLine(); }
```

```
            $null = $sb.Append($_)
```

```

    }

    }

    end { [windows.forms.clipboard]::SetText($sb.ToString()) }
}

function Get-Clipboard {

    [windows.forms.clipboard]::GetText()

}

# funkce clip zastupuje Get-Clipboard i Set-Clipboard podle kontextu:
# gc c:\test.txt | clip
# clip | sc c:\test.txt

function clip {

    param([Parameter(Mandatory=$false, ValueFromPipeline=$true)][object]
    $s)

    begin { $sb = new-object Text.StringBuilder }

    process {

        $s | % {

            if ($sb.Length -gt 0) { $null = $sb.AppendLine(); }

            $null = $sb.Append($_)

        }

    }

    end {

        if ($sb.Length -gt 0) { $sb.ToString() | Set-Clipboard }

        else { Get-Clipboard }

    }

}

```

K tomu, aby přístup ke schránce fungoval, je nutné, aby PowerShell běžel s přepínačem -sta, nebo abyste tento kód používali v ISE prostředí. Pokud byste přesto chtěli mít přístup ke schránce i v MTA režimu, podívejte se na [řešení](#).

Je to všechno?

Co by si zasloužilo alespoň zmínit, je cmdlet `New-WebServiceProxy`, dále práce s sql serverem (pomocí .NET prostředků), vytváření jednodušších GUI aplikací (ať už WinForms,

nebo WPF). Tato témata se už do přehledu nevešla vzhledem k jejich rozsahu. Na internetu ovšem je k nalezení plno zdrojů, které zájemcům pomohou.

Za sebou mám základní věci, které vás jako vývojáře mohou uchvátit, nebo nechat chladným. Stále platí, že nejdůležitější je první bod – rychlé a efektivní řešení běžných úkolů a v tom se PowerShellu těžko něco vyrovná.

PS A ACTIVE DIRECTORY

Dnes podíváme na oblast, která možná zajímá většinu z vás – administraci Active Directory pomocí PowerShellu.

1.1 Možnosti

Nejprve se podívejme, jaké možnosti máme. Jako první můžeme spustit následující příkaz

```
PS C:\> admgmt.msc
```

OK, beru zpět, tuto metodu asi nemůžeme považovat za úplně PowerShell-like. Použitím některé z následujících technik už jsme na tom ale v konzoli PowerShellu lépe.

- ADSI
- .NET pomocí SDSP
- Quest cmdlets
- Windows 2008 R2 cmdlety nebo pomocí [Active Directory Management Gateway Service](#) pro nižší verze Windows serveru.

Já budu většinu příkladů ukazovat na cmdletech od firmy Quest. Mám k tomu dva hlavní důvody.

1. Přístup přes ADSI mi přijde moc VBS-like. Nutno říci, že to není špatná cesta – pro některé z vás možná ta jediná použitelná (pokud mate například zakázáno instalovat nástroje třetích stran), nicméně na ADSI bylo v minulosti opravdu tolik příkladů, že by to bylo nošením dříví do lesa (krátkému srovnání se ale stejně budu věnovat). Pokud vás přístup přes ADSI bude zajímat, podívejte se na konec článku, kde uvádím odkaz na jednu moc zajímavou knihu.
2. Nativní cmdlety (pro Windows 2008 R2) nejsou ještě tolik rozšířené. Přecijen R2 server tu s námi není tak dlouho a i když rozhodně předpokládám v budoucnu opravdu široké použití, zatím to prostě není ono.

1.2 Základní práce s AD

Typickým použitím PowerShellu při přístupu do AD je dotazování na uživatele, skupiny, členství ve skupinách, atd. Oproti AD konzoli máme výhodu v tom, že v konzoli množství dotazů ani nejsme schopni (při rozumném čase a úsilí) splnit. Zkuste například spočítat (a následně zobrazit v koláčovém grafu) množství lidí v konkrétních odděleních (organizačních jednotkách). Bez použití PowerShellu (nebo nástrojů třetích stran) máte práci asi tak na hodinu, s PowerShelllem vyřešíte tento úkol za pět minut a zbytek oné hodiny se můžete koukat na slidy z poslední [TechNet conference](#).

Poznámka: Cmdlety od firmy Quest jsou dostupné [volně ke stažení](#). Nainstalujte je klasickým způsobem a poté je přidáte do aktuální konzole PowerShellu pomocí příkazu

```
PS C:\> Add-PSSnapIn Quest.*
```

Pokud je máte správně nainstalované, měl by vám následující příkaz vrátit všechny cmdlety pro práci s AD.

```
PS C:\> Get-Command -Module Quest.*
```

Pojďme si rovnou ukázat příklad z praxe:

```
PS C:\> Get-QADUser | Group-Object ParentContainer | Sort-Object Count -Descending | Select-Object Count, Name -First 5 | Out-Chart -xField 'Name' -yField 'Count'
```

Tímto příkazem zjistíme pět oddělení s největším počtem zaměstnanců a zobrazíme je v grafu. **Out-Chart** není standardním příkazem PowerShellu a můžete si jej [stáhnout ze stránek Chada Millera](#). Jako první provedeme pomocí **Get-QADUser** dotaz na všechny uživatele, pak je sdružíme podle organizační jednotky, seřadíme podle počtu a následně zobrazíme. Většinu cmdletů znáte z předchozích dílů a nyní vše skládáme dohromady pomocí rour v jeden funkční celek.

Dalším typickým dotazem do AD je zjišťování uživatelů se zamčenými nebo disablovanými účty.

```
PS C:\> Get-QADUser -SearchRoot domain.cz/UserOU -Locked
```

```
PS C:\> Get-QADUser -Disabled
```

Naprosto jednoduchý přístup, který přebírá vše dobré, co jste se již v PowerShellu naučili. Pokud chcete zamčené uživatele odemknout, stačí přidat rouru a další cmdlet a máte zase všechny uživatele zpět odemčené.

```
PS C:\> Get-QADUser -Locked | Unlock-QADUser
```

Jistě jste si všimli, že použité cmdlety mají strukturu **<Verb>-QAD<Noun>**. Firma Quest správně tušila, že i Microsoft přijde se svými cmdlety a přidáním prefixu QAD se vyhnula případným budoucím konfliktům. Na Windows 2008 R2 opravdu existuje cmdlet [Get-ADUser](#).

Další zadání může být například následující. “Dokážeme zjistit, kolik jsme v loňském roce vytvářeli nových účtů?” Toto se vám může hodit v případě plánování zdrojů a alespoň přibližně můžete odhadnout vaše následné náklady. Řešením je tento krátký one-liner. Termínem one-liner se rozumí příkaz PowerShellu, který napíšete na jednu řádku. Už jsem ale viděl i one-linery na několik (desítek) řádek, pouze vhodně spojené rourou.

```
PS C:\> Get-QADUser -CreatedAfter "1/1/2009 00:00" -SearchRoot 'domena/czOU'
```

Nutno podotknout, že účty po odchodu lidí nemažu, ale nechávám je určitou dobu ve speciální organizační jednotce, což se mi pro tento případ hodilo. Pro rozdělení na jednotlivá oddělení bych postupoval stejně jako v prvním příkladě.

1.3 Vytvoření nového uživatele

Pojďme si porovnat vytvoření nového uživatele pomocí ADSI a Quest cmdletů. Pokud jste někdy vytvářeli účty ve VBS, asi jste právě ADSI použili. Proto pro vás nebude následující příklad žádným překvapením:

```
PS C:\> $OU = [ADSI]"LDAP://ou=czOU,dc=domain,dc=cz"
PS C:\> $User = $OU.Create("user", "cn=DavidM")
PS C:\> $User.Put("sAMAccountName", "davidm")
PS C:\> $User.SetInfo()
```

Pomocí Quest cmdletů vytvoříme uživatele následujícím způsobem:

```
PS C:\> New-QADUser -samAccountName 'davidm' -name 'DavidM' -
ParentContainer 'OU=czOU,DC=domain,DC=cz'
```

Ve Windows 2008 R2 můžete použít následující příkaz:

```
PS C:\> New-ADUser -SamAccountName "davidm" -DisplayName "DavidM" -Path
'OU=czOU,DC=domain,DC=cz'
```

Při vytváření jednoho uživatele asi výhody konzole nepocítíte, ale představte si nástupy nových lidí každý rok a čas strávený s přípravou PowerShellí verze se vám vyplatí. Nebudu zde ukazovat použití při načítání nových uživatelů z CSV souboru. Takových příkladů existují na webu desítky. Odkáží vás ale na TechNet Magazine, kde se právě tomuto úkolu [věnoval v několika pokračováních](#) Don Jones.

1.4 Práce se skupinami

Pokud vytvoříte nového uživatele, musíte (měli byste) ho také přiřadit do skupiny. Opět se jedná o jednoduchý příkaz.

```
PS C:\> Add-QADGroupMember -identity cz\ProxyServerAccess -member davidm
```

Pokud potřebujete později zjistit všechny uživatele konkrétní skupiny, pomůže vám tento cmdlet:

```
PS C:\> Get-QADGroupMember cz\ProxyServerAccess
```

Práci se skupinami a vytváření uživatele můžete také spojit do jednoho příkazu

```
New-QADUser | Add-QADGroupMember
```

Úplně stejně můžete pracovat i s počítači. Asi dokážete odhadnout jméno cmdletu pro práci s nimi.

```
PS C:\> Get-QADComputer -SearchRoot 'domain.cz/Domain Controllers'
```

Pokud budete některé cmdlety používat častěji a nehodí se vám jejich základní parametry, můžete si vytvořit vlastní funkci, do které tento cmdlet “zapouzdříte” (o funkcích si možná povíme něco později). Na mém počítači mohu například zjistit počet mnou upravených AD cmdletů takto:

```
PS C:\> (Get-Command *-DMAD* | Measure-Object).Count
12
```

1.5 Další informace

Dnešní téma jsme vzali trochu “letem, světem”. Důvod je jednoduchý. Při práci s Active Directory je dobré si uvědomit jednu základní věc. Všechny potřebné cmdlety jsou vám k dispozici a je pouze na vás, jak je budete kombinovat dohromady. Vždy je dobré znát základy PowerShellu (cmdlety, které jsem ukazoval v předchozích dílech) a vše nové si přizpůsobit pro vlastní potřebu.

O PowerShellu a Active Directory bylo napsáno už mnoho článků. Nechtěl jsem zde opakovat některé často používané konstrukce. Pokud vás ale tento článek trochu vtáhl do tématu a rádi byste se dozvěděli více, uvádím zde několik zajímavých odkazů

- O tématu administrace AD pomocí PowerShellu vyšla celá kniha. Vydalo ji nakladatelství Sapien Press a jmenuje se [Managing Active Directory with Windows PowerShell: TFM](#). Pokud ji chcete dostat zdarma, přihlaste se na [těchto stránkách](#). Jedná se opravdu o velký zdroj informací o tomto tématu.
- V PowerShellu můžete například vytvořit testovací doménu se všemi potřebnými objekty. Stačí použít [skript](#) publikovaný na [PoshCode.org](#). K prostudování doporučuji i [originální skript Dmitriho Sotnikova](#).
- [Dmitry Sotnikov](#) je vůbec cenným zdrojem práce s Active Directory. Další zajímavostí je například [porovnání snapshotů stavu AD](#).
- Jestli vás více zajímají [nativní cmdlety W2K8 R2](#), podívejte se na TechNet.
- Posledním odkazem je pro dnešek [blog Active Directory PowerShell týmu](#).

TIPY A TRIKY

Při práci v PowerShellu, stejně jako v jakémkoliv jiném programovacím či skriptovacím jazyku, narazíte čas od času na nějakou zajímavost, nebo obecný vzorec, který vám může usnadnit práci, zpřehlednit ji, nebo vám konečně pomůže pochopit nějaký obecný princip. V tomto článku vám předložím některé z těchto tipů a zajímavostí. Věřte ale, že je to jen malá část.

Navíc se snažím nerozebírat pokročilá témata, ale ukázat především tipy základní. Doufám, že alespoň některé z nich pro vás budou nové.

Konzole

Libovolná jména funkcí

PowerShell se používá dvěma různými způsoby: většinu kódu máme ve svých skriptech a tyto pouštíme podle potřeby. Anebo máme otevřenou konzoli a píšeme ps kód a spouštíme bez ukládání.

V prvním případě je velmi důležitá přehlednost kódu, jeho srozumitelnost a pochopitelnost. Proto se doporučuje používat plná jména cmdletů, ne jen `gci` ale `Get-ChildItem` (nehledě na to, že na jiném systému může `gci` být aliasem pro něco úplně jiného).

Ve druhém případě ovšem používáme hojně aliasů (`gci|?{!$_.PSIsContainer}|select -exp Length`) a co nejkratších konstrukcí. Důležité je dosáhnout cíle, ale téměř nezáleží na tom *jak*. Díky tomu, že PowerShell je velmi benevolentní, umožní nám pojmenovat funkce a aliasy velmi zajímavými jmény. Podívejte se na příklady.

```
PS> # dobře známé funkce ?? a ?:
```

```
PS> function ?? { if ($args[0]) { $args[0] } else { $args[1] } }
```

```
PS> function ?: { if (&$args[0]) { $args[1] } else { $args[2] } }
```

```
PS> ?? $null 'default value'
```

default value

```
PS> ?: {1} 'is 1' 'is not 1'
```

is 1

```
PS> ?: {get-process nonexistent -ea 0} 'process exists' 'process doesn't exist'
```

process doesn't exist

```
PS> function \ { 'this is slash' }
```

```
PS> function * { '*'*10 }
```

```
PS> # definuje funkci, jejíž jméno je CTRL+D
```

```
PS> New-Item -Path "function:$([char][int]4)" -ItemType function -Value  
{ write-host 'CTRL+D!' }
```

I když PowerShell nechává na nás, jak se budou naše funkce jmenovat, dělejme tak s rozmyslem.

[Zápis bytů](#)

Nejlépe a okamžitě vše bude vidět na příkladu

```
PS> 1kb, 1mb, 1gb, 1tb, 1pb
```

1024

1048576

1073741824

1099511627776

1125899906842624

Na konec číselné hodnoty můžete přidat jednotku (v bytech). PowerShell toto automaticky vyhodnotí za vás. V důsledku pak zjednodušíte kód např. takto:

```
gci | ? { !$_.PSIsContainer -and $_.Length -gt 15mb }
```

Operátory a proměnné

Řetězení operátorů

Tato technika není příliš často používána, ale v mnoha případech dokáže nahradit použití cmdletů a pipeline. Pro demonstraci předpokládejme, že náš adresář obsahuje tyto soubory:

```
ch01-2010-03-01.txt
ch01-2010-03-02.txt
ch02-2010-03-01.txt
ch02-2010-03-02.txt
ch03-2010-03-01.txt
ch04-2010-03-01.txt
```

My z nich chceme vyfiltrovat jen ty, které začínají *ch01*. Z nich pak chceme získat pouze střední část zkonvertovatelnou na `[datetime]`. Tradičně bychom toto mohli provést přibližně takto:

```
Get-ChildItem c:\temp\aa\ | select -exp Name | ? { $_ -like 'ch01*' } |
% { $_ -replace 'ch01-|\.txt', '' }
```

Ale stejně tak můžeme použít i následující způsob. Všimněte si, že část `select -exp Name` v tomto případě není nutná, protože dojde ke konverzi `[FileInfo]` na `string`, při níž se bere jméno souboru.

```
(Get-ChildItem c:\temp\aa\ | select -exp Name) -like 'ch01*'
-replace 'ch01-|\.txt', ''
```

Zde jsme použili dvou vlastností:

Zprvé – operátory se dají řetězit. Tedy i operátor `replace` by někdo kvůli čitelnosti mohli rozdělit na dva. Výsledek by pak vypadal takto: `...-replace 'ch01-', '' -replace '\.txt', ''`. Výsledek předchozího operátoru se uplatní při vyhodnocování následujícího operátoru. Toto jsem neviděl pořádně zdokumentované, tedy se odvolávám pouze na své dosavadní zkušenosti.

Zadruhé – některé operátory pracují nejen nad skalárními hodnotami, ale také nad poli. Proto jsme mohli operátorům `like` a `replace` jako levý operand předat pole a vrátila se nám korektní hodnota. V případě, že bychom ale uvedený příklad chtěli dovést až do konce a hodnoty `2010-03-02` a podobné chtěli převést na `datetime`, pak toto je špatně: `... -replace 'ch01-|\.txt', '' -as [datetime]`. V tomto případě se operátor snaží převést vstupní objekt (pole) na čas a toto pochopitelně nedopadne dobře. Stačí ale drobná změna a máme funkční kód.

Srovnajte:

```
Get-ChildItem c:\temp\aa\ |
```

```

select -exp Name |
? { $_ -like 'ch01*' } |
% { $_ -replace 'ch01-|\.txt','' } |
% { $_ -as [datetime] } |

? { $_ -le '2010-03-01' }
(Get-ChildItem c:\temp\aa\ `

-like 'ch01*' `

-replace 'ch01-|\.txt','' `

-as [datetime[]] `

-le '2010-03-01'

```

Automatické proměnné \$\$, \$^

\$^ a \$\$ jsou automatické proměnné, které mají smysl hlavně při interaktivní práci v shellu. Ne vždy je použijete, ale hodí se je znát. O co jde, je nejlépe vidět na příkladu:

```

PS> Get-ChildItem -rec c:\temp\powershelltest\version1

...výpis souborů

PS> $^

Get-ChildItem

PS> $$

c:\temp\powershelltest\version1

```

Proměnné obsahují první a poslední token na předchozím řádku. Přesněji: *Contains the first token in the last line received by the session* a *Contains the last token in the last line received by the session*.

Mohou vám tak ušetřit hlavně psaní dlouhých cest do příkazů jako je `Get-ChildItem`, `Get-Item` atd. Otázka na StackOverflow dokládá, že se najdou tací, kteří tuto proměnnou skutečně [používají](#).

Objekty

Objekty a typy

Pro mě nejužitečnější tip k práci s objekty a typy je přetypování. Nemám na mysli operátory, ale jednodušší způsob, jak využít jednoparametrický konstruktor. Příklady napoví.

```

PS> Add-Type @"

using System;

using System.Net;

```

```

public class Test1 {

    public Test1(string s) { Console.WriteLine("Test1 - string ctor: {0}",
s); }

    public Test1(int i) { Console.WriteLine("Test1 - int ctor: {0}", i); }

    public Test1(WebClient c) { Console.WriteLine("Test1 - webclient   ctor:
{0}", c); }

}

public class Test2 {

    public Test2(string s1, string s2) { Console.WriteLine("Test1 -
string ctor: {0}, {1}", s1, s2); }

}

```

"@

```

PS> [Test1] 'a' > $null                                # použije 1. konstruktor

PS> [Test1] 1 > $null                                   # použije 2. konstruktor

PS> [Test1] (New-Object Net.WebClient) > $null # použije 3. konstruktor

PS> [Test1] (date) > $null                               # chyba - Multiple ambiguous
overloads...

PS> [Test2] @('a','b') > $null                           # bohužel nejde
PS>
[System.Net.IPAddress[]]'127.0.0.1','192.168.45.1','192.168.45.2','192.168.
45.3'

```

Je tedy možné, pokud máme k dispozici jednoparametrický konstruktor, vyhnout se cmdletu `New-Object` a pouze použít přetypování.

Při vytváření instancí nějakého .NET typu je potřeba vždy uvádět plnou cestu. Například `New-Object System.Text.StringBuilder`. Co nám může pomoci?

Předně není nutné psát `System`. Kratší zápis vypadá takto:

```

PS> $sb = new-object Text.StringBuilder

```

Pokud máme vytvářet více instancí z jednoho namespace a nechceme pořád daný namespace opisovat, můžeme si jej uložit do proměnné:

```

PS> $g = 'System.Collections.Generic.'
PS> $list = New-Object ($g + 'List[int]')

```

Skládání stringů možná nikoho nepřekvapilo a možná to každému přišlo naprosto přirozené. Méně přirozeně může vypadat poslední tip k typům. Odkaz na třídu si můžeme uložit do proměnné a později použít hlavně při volání statických metod.

```
PS> Add-Type -assembly system.windows.forms #nacteme windows.forms

PS> $forms = [System.Windows.Forms.MessageBox]

PS> $forms::Show('Hello')
```

Přidávání properties k PSObject

Za časů PowerShell V1 se k instancím obecného objektu přidávaly *property* pomocí cmdletu `Add-Member`.

```
$info = new-object PSObject |

    Add-Member NoteProperty App 'ap' -pass |

    Add-Member NoteProperty Account 'account' -pass |

    Add-Member NoteProperty StatusId 1000 -pass
```

Někteří si život usnadňovali trikem pomocí `Select-Object`.

```
$info = '' | Select-Object App,Account,StatusId

$info.App,$info.Account,$info.StatusId = 'app','account',1000
```

V PowerShellu V2 ale máme k dispozici nový parametr `-property`, který zápis velmi usnadní a zpřehlední.

Už nikdy víc `Add-Member` a `Select-Object` jen kvůli přidání property!

```
$info = new-object PSObject -property @{App='app'; Account='account';
StatusId=1000 }
```

Práce s hashtable

Pro nejjednodušší případy, kdy i vyrábění PSObjectu je pro nás zdlouhavé, můžeme použít hashtable. Přistupovat do ní totiž můžeme nejen pomocí hranatých závorek, ale i pomocí tečkové notace.

```
PS> $h = @{Height=180; Weigh=80; Name='El'; Surname='Hombre' }

PS> $h.Name

El
```

Někdy může být zajímavé i použití více hodnot do indexu:

```
PS> $h['Height', 'Name']
```

180

E1

Tento přístup má ale i své nevýhody: Při přístupu přes tečkovou notaci nefunguje doplňování pomocí *[TAB]*. Navíc se tyto "objekty" nedají třídit podle dané "property".

```
1..10 | % { New-Object PSObject -property @{Num=$_} } | Sort Num -desc  
#funguje
```

```
1..10 | % { @{Num=$_} } | Sort Num -desc #nedava spravne vysledky
```

Cmdlety

Splatting

V PowerShellu V2 máme k dispozici splatting operátor @. Používá se při práci s parametry funkcí a cmdletů. Za běhu si můžeme určit, které parametry budeme chtít předat.

```
PS> function Write-Parameters {  
    param([string]$p1, [int]$p2, [switch]$p3, [string]$p4)  
    Write-Host P1: $p1  
    Write-Host P2: $p2  
    Write-Host P3: $p3  
    Write-Host P4: $p4  
}
```

```
PS> $parameters1 = @('2000') # pole
```

```
PS> $parameters2 = @('2. polozka v poli') # pole
```

```
PS> $switch = @{p3=$true; p1='P1' } # hashtable
```

```
PS> Write-Parameters @parameters1 @parameters2 @switch
```

```
PS> Write-Parameters 2000 '2.položka v poli' -p3 -p1 P1
```

Všimněte si, že výstup z volání `Write-Parameters` je stejný v obou případech. Za jméno funkce/cmdletu můžeme předat pole, nebo hashtable, které určují vstupní parametry. Pokud předáme pole, účinek je podobný, jako bychom zapisovali pouze argumenty a spoléhali se na navázání na parametry pomocí pozice.

Pokud předáme hashtable, argumenty jsou navázány stejně, jakobychom před každým z nich specifikovali jméno parametru.

Pozn.: vyhodnocování parametrů je relativně složitý proces. Více je popsáno v knize [Windows PowerShell in Action](#). Rád bych jen upozornil, že v našem případě se nejdříve navážou argumenty určené explicitně, tj. před nimi je jméno parametru (např. `-p1 P1`). A až teprve poté se navazují zbylé argumenty pomocí pozice na dosud nenavázané parametry. V našem případě se tedy nejdříve naváže parametr *p1* a *p3*. Teprve potom se začíná navazovat argument *2000*. Parametr *p1* už je navázaný, tedy je přeskočen. První volný je parametr *p2*, proto se hodnota naváže na něj. To stejné platí pro hodnotu *2. polozka v poli*.

Splatting použijeme tehdy, pokud chceme využít už existujících cmdletů, jako je např. `Get-ChildItem`.

```
PS> function Get-MyItems {  
  
    param([switch]$all, [string]$extension, [string]$directory)  
  
    $p = @{}  
  
    if ($all) { $p['recurse'] = $true }  
  
    if ($extension) { $p['filter'] = ".*$extension" }  
  
    if ($directory) { $p['literalPath'] = $directory }  
  
    Get-ChildItem @p  
  
}
```

```
PS> Get-MyItems -extension txt
```

```
PS> Get-MyItems -all -directory G:\temp\blog
```

[Alias = Get-Alias](#)

Narazili jste někdy na použití např. `alias gm` a divili se, co je to vlastně ten `alias` zač? V seznamu funkcí se nenachází, stejně tak v seznamu aliasů a cmdletů. Podobně funguje `date`, `job`, `childitem`, `item`, `verb`, `service`, atd.

Pokud PowerShell zjistí, že neexistuje daný příkaz (např. můžeme mít definovanou funkci `service`), zjistí, zda existuje alias `Get-...`. Pokud alias existuje, zavolá jej. Pokud neexistuje, zkusí podobně existenci funkce a cmdletu a případně existující příkaz zavolá.

[Vyhodnocení proměnné/výrazu v řetězci](#).

To, že se proměnné v řetězci vyhodnocují, ví asi každý, kdo s PowerShellem pracuje.

Pokud jde o proměnnou, která obsahuje pole hodnot, pak tyto hodnoty jsou spojeny pomocí speciální proměnné `$ofs`. Pokud bychom chtěli hodnoty oddělit pomocí svislítek, můžeme použít toto:

```
PS> $ofs = "|"; $pole = 1,2,3,4; "$pole"
```

```
1|2|3|4
```

Mnohem pěknější je ovšem v tomto případě použít operátor `-join` než nějakou magickou proměnnou.

```
PS> $pole = 1,2,3,4; $pole -join "|"
```

```
1|2|3|4
```

Pokud chceme přistupovat k property objektu uložené v proměnné, musíme už použít závorek:

```
PS> $pole = 1,2,3,4; "velikost: $($pole.Length)"
```

```
velikost: 4
```

Při vyhodnocování výrazu nejsme omezeni, tj. můžeme použít např. indexaci, můžeme vytvořit objekt, atd. Tato praktika se ovšem nedá obecně doporučit. Výraz lze téměř jistě vždy vyhodnotit dříve než až při použití v řetězci:

```
PS> "$((new-object  
net.webclient).DownloadString('http://google.com').Substring(0, 50))"
```

```
<!doctype html><html><head><meta http-equiv="conte
```

Pokud se ovšem seznam vnořených property dozvíme až za běhu, můžeme použít chytré metody `ExpandString`:

```
PS> $x=[xml]'<a><b><c>some value</c><c>some second value</c></b></a>'
```

```
PS> $prop='a.b.c[1]' #tento řetězec zjistíme až za běhu
```

```
PS> $stringToExpand = "`$($x.$prop)"
```

```
PS> $ExecutionContext.InvokeCommand.ExpandString($stringToExpand)
```

Parametr `-OutputVariable`

`-OutputVariable` je parametr společný všem cmdletům. Používá se v situacích, kdy cmdlet vrací nějaké objekty (tj. u `Write-Host` nemá smysl) a chceme zachytit výstup z daného cmdletu do proměnné. Může se nám např. hodit při debugování nebo na zjednodušení skriptů. V příkladech budu používat alias `ov`.

```
PS> Get-Process svchost -OV svchosts | ? { $_.Handles -gt 500 } -OV  
svchosts2
```

```
PS> $svchosts.Count
```

15

```
PS> $svchosts2.Count
```

4

Tento parametr funguje podobně jako cmdlet `Tee-Object`, který má mimo jiné také možnost uložit obsah do proměnné, ale jednak je použití parametru jednodušší, druhak je možné do proměnné přidávat předřazením znaménka `+`.

```
PS> Get-Process svchost -OV procs; Get-Process firefox -OV +procs; Get-Process powershell -OV +procs
```

```
PS> $procs | select -exp ProcessName -unique
```

svchost

firefox

powershell

Ostatní Krátce

- `Start-Transcript` spustí "logování" toho, co se děje ve vaší konzoli. Hodí se především tehdy, když experimentujete s neznámým API – dodatečně se dají příkazy a výstupy prohlédnout a jednoduše najít, která cesta vedla k cíli. Na ukončení logování pak použijete cmdlet `Stop-Transcript`.
- `ii` adresář otevře adresář ve Windows Exploreru. `ii soubor` spustí default akci nad souborem.
- `gci env`: vylistuje systémové proměnné.
- Jestli vám někdy chyběl příkaz `Stop-Pipeline`, podívejte se na článek [Cancelling a Pipeline](#).
- Pokud potřebujete z kolekce i odebírat, můžete použít `ArrayList`, nebo za určitých okolností využít i přiřazování do více proměnných. Příklad napoví: `$pole = 1..10; while($pole) { $p,$pole = $pole; write-host $p }`.
- Některé cmdlety mají dynamické parametry. Pokud se chcete na tyto parametry podívat podrobněji, doporučuji [About Dynamic Parameters](#).

- Zkuste v konzoli napsat na nový řádek # a mačkat *TAB*. PowerShell vám bude vypisovat příkazy z historie, počínaje posledním. Je to lepší varianta pohybu pomocí šipek. V tomto případě se totiž nepohybuje po řádcích, ale po celých příkazech, které se roztáhnou přes více řádků. Když pak zkusíte napsat #<řetězec obsažený v nějakém předchozím příkazu>[*TAB*], bude nápověda cyklit jen po příkazech, které obsahují daný řetězec.
- A ještě jeden tip na doplňování pomocí *TAB*. V některých situacích si nepamatujete přesně jméno souboru, ale víte nějaký řetězec, který jeho jméno obsahuje, případně příponu. Pak stačí výraz zapsat pomocí wildcards `c:\temp\log??iis*` a zmáchnout [*TAB*]. Konzole bude nabízet jen soubory odpovídající wildcards.
- *Windows API*

Možná je to všem programátorům zřejmé, možná ne, ale – v PowerShellu můžete pomocí `Add-Type` používat i Windows API. [Joel Bennet](#) má na [PoshCode.org](#) krásnou [ukázkou](#). Jeho modul umožní skrýt a zobrazit okna, nebo nastavit průhlednost oka. Použití je velmi jednoduché.

```
PS> $n = Select-Window | ? {$_ .title -match 'untitled' } #vybere
instanci Notepadu
```

```
PS> $n | Set-GhostWindow -Percent 50 #nastav průhlednost na 50%
```

```
PS> $n | Remove-GhostWindow # odstraň průhlednost
```

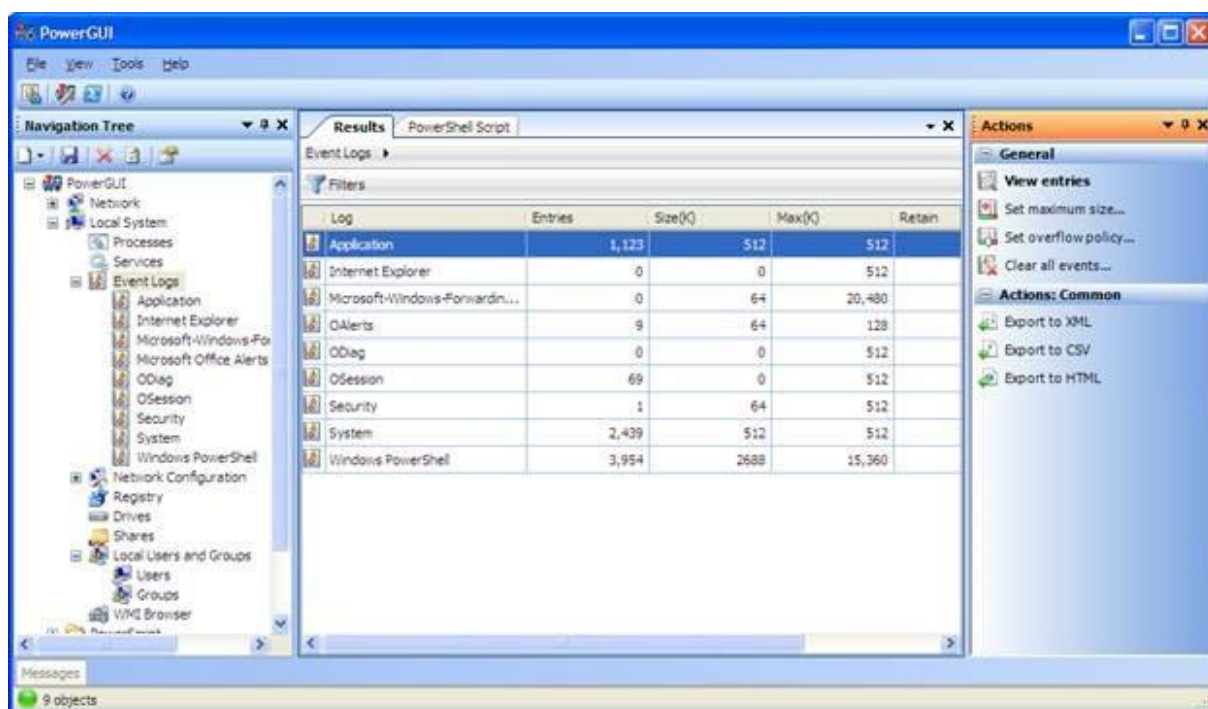
Pokud máte své vlastní tipy, které vám přijdou zajímavé, neváhejte a využijte komentářů. Ostatní čtenáři je jistě ocení!

NADSTAVBY POWERSHELLU

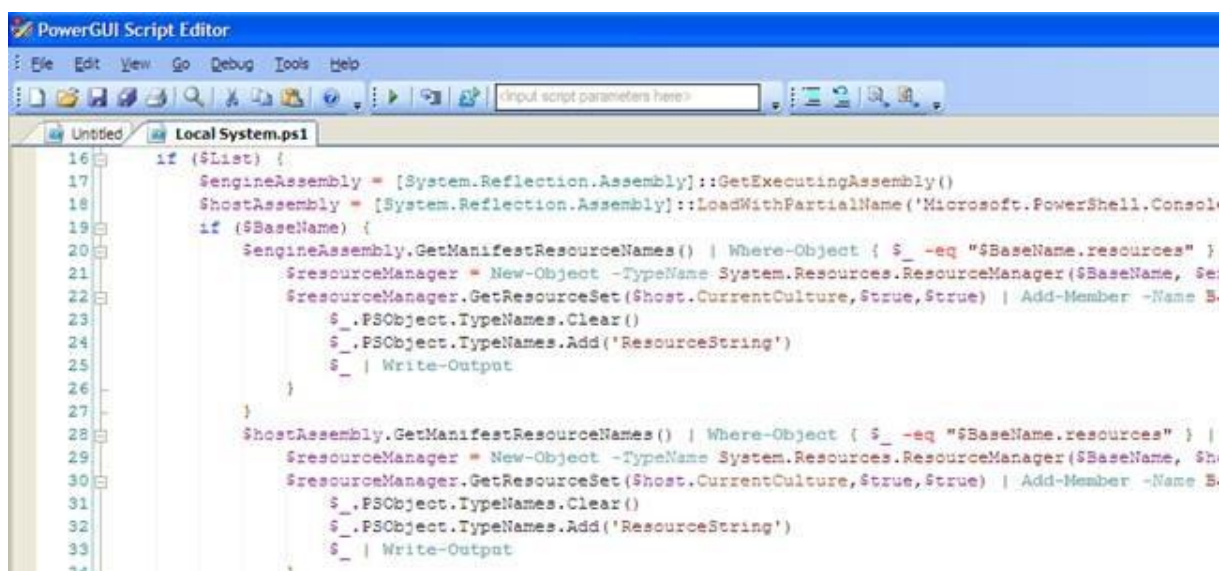
Dnes se podíváme mimo standardní cmdlety dodávané přímo s PowerShellem a řekneme si něco o zdarma dostupných rozšířeních. Zároveň si ukážeme zdroje, které je dobré sledovat.

PowerGUI

[PowerGUI](#) je možná nejznámější, zdarma šířený, nástroj pro PowerShell. Obsahuje administrátorskou konzoli a editor skriptů.



Administrátorská konzole je podobná MMC konzoli (v3) a skládá se ze tří hlavních částí – stromové struktury, hlavního panelu a panelu akcí. Konzoli je možno rozšířit pomocí takzvaných management packů (analogie se snap-iny v MMC) a v současné době těchto management packů existuje několik desítek. Dostupné jsou na adrese [\[\[www\]\].](http://www.microsoft.com/technet/scriptcenter/PowerGUI.mspx) Celý management pack je vlastně PowerShell skript, zabalený ve specifickém XML souboru (PowerGUI pro vás tento XML soubor generuje automaticky).



Druhou částí PowerGUI je editor skriptů. Dle mého názoru se jedná o velice zdařilý nástroj a pokud píšete skripty, měli byste mu určitě věnovat pozornost (a porovnat, zda vám nebude vyhovovat lépe, než standardně dodávané PowerShell ISE). Editor obsahuje všechny základní vlastnosti, které si můžeme jako administrátoři přát (vývojářům možná budou některé části chybět). Funguje zde Intelli-Sense (doplňování např. Vlastností a metod objektů, doplňování jmen proměnných, ...), obsahuje možnost vkládání Snippetů (připravené složitější konstrukce), pěkně funguje debugování, zvýraznění syntaxe nebo například šikovný export skriptu do HTML, které pak můžete umístit na své stránky.

Před několika málo měsíci uvedla firma Quest na trh také PowerGUI Pro. To obsahuje navíc možnost napojení na některý ze softwarů pro verzování a také možnost běhu PowerShellu přes webové rozhraní.

PSCX

[PowerShell Community Extensions](#) – jeden z prvních projektů, který před několika lety rozšířil možnosti PowerShellu v1 o velice zajímavé skripty. V současné době obsahuje 149 skriptů, například pro práci s archivy, HTML/XML nebo některé více „programátorské“, např. Invoke-GC. V současné době jsou dostupné dvě verze: produkční 1.2 dostupná jako PSSnapin, kompatibilní i s PS v1 a verze 2.0 Beta 2, která je dostupná pouze pro verzi 2 a instaluje se z modulu.

SQLPSX

[SQL Server PowerShell Extensions](#) – projekt, za kterým stojí hlavně Chad Miller. Obsahuje množství skriptů pro správu SQL serveru a pokud vám nestačí dva cmdlety dodávané se SQL serverem, zde si určitě vyberete. V poslední verzi byl do SQLPSX přidán modul pro práci v ISE (sqlise) a chystá se i database object browser (pak už opravdu nebudu potřebovat SSMS J U sqlise bych se rád na chvílku zastavil po importování tohoto modulu se v menu objeví dvě nové položky: ISE Pack (o něm až za chvílku) a SQL Ise.



Pokud potřebujete udělat dotaz do vaší databáze, mohli byste postupovat třeba takto:

- V ISE nainportujte modul: ipmo sqlise. Uvidíte, že se vám změní prompt.
- Připojte se na danou databázi.
- V editoru zadejte jednoduché query (všimněte si, že jej schválně zadávám neformátovaný)
- Pomocí klávesové zkratky CTRL+4 nechte sqlise váš kód naformátovat.

```
PS C:\Scripts#[CONNECTED][cz 01.SMS_]: >
```

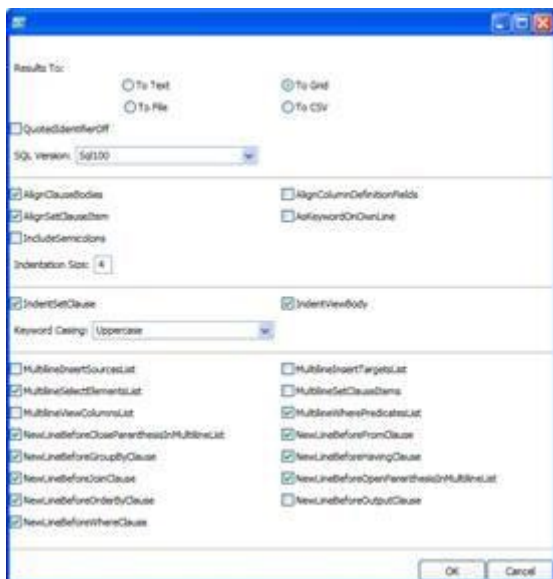
```
select * from dbo.v_R_System
```

```
SELECT *  
FROM   dbo.v_R_System;
```

- Stiskněte CTRL+ALT+F5 a tím dotaz spustíte. Po chvíli se vám objeví výstup v GridView. Title GridView je stejný jako jsou souboru v ISE – v mém případě tedy Untitled.ps1.

ResourceID	ResourceType	Active0	ACSite_Name0	Client0	Client1
10 522	S	1		1	1
11 472	S	1		1	1
12 892	S	1		1	1
13 028	S	1		1	1
13 205	S	1		1	1
13 245	S	1		1	1
13 726	S	1		1	1
13 730	S	1		1	1
13 736	S	1		1	1
13 740	S	1		1	1
13 791	S	1		1	1

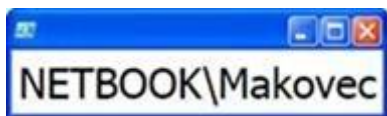
Vzhledem k tomu, že celý isepack je psaný v PowerShellu není problém si jej upravit pro své potřeby. Některé základní možnosti konfigurace jsou dostupné přes menu Add-ons/SQLIse/Options



PowerShell Pack – WPK

S Windows 7 byl uvolněn také Resource Kit, který obsahuje jednu zajímavou část – [Windows PowerShell Pack](#). Tento balík se skládá z několika modulů z nichž nejzajímavější je bezesporu WPK (Windows Presentation Foundation (WPF) PowerShell Pack) – sada skriptů pro tvorbu grafického rozhraní. Obsahuje 716 různých skriptů a jeho možnosti jsou prostě ohromné. S trochou trpělivosti můžete vytvořit jakékoli GUI. Například lehká obměna pro [10. úkol letošních Scripting Games](#) by mohla vypadat takto:

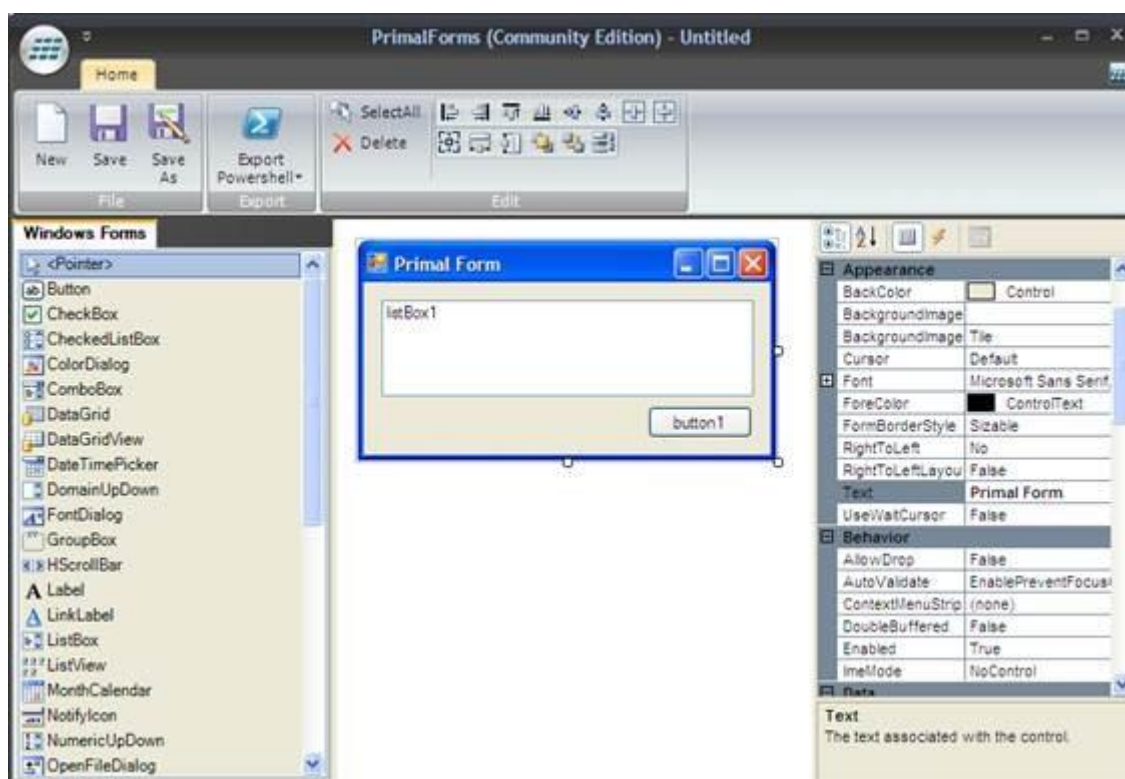
```
New-Label "$((gwmi win32_computersystem).UserName)" -FontSize 30 -Show
```



Dalším nástrojem pro tvorbu grafických rozhraní jsou např. také [PowerBoots](#) od [Joela Bennetta](#). I když jsou oba projekty velmi pěkné, pro někoho může být překážkou, že vytváření grafických komponent není úplně intuitivní. V tomto případě se může hodně hodit následující nástroj.

PrimalForms Community Edition

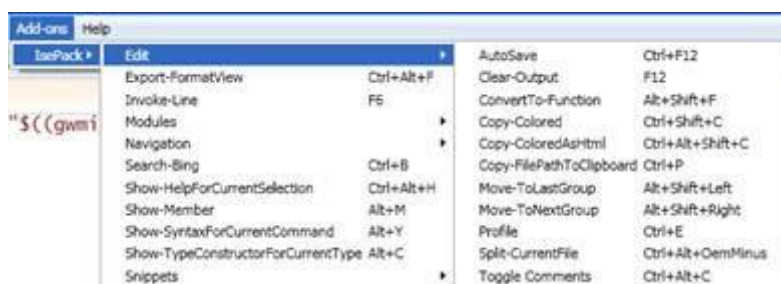
Pokud vám psaní GUI z příkazové řádky přijde nepohodlné (čemuž bych se opravdu nedivil), můžete použít [PrimalForms Community Edition](#) – další zdarma dostupný nástroj, tentokrát od firmy Sapient (nejznámější je zřejmě jejich nástroj PrimalScript, který se používá ve všech skriptovacích MOC kurzech).



Pomocí přetažení prvků z levého seznamu do formuláře si můžete navrhnout design a PrimalForms potom automaticky vygenerují kód. Do tohoto kódu poté pouze doplníte obsluhu jednotlivých prvků. Pokud jste někdy uvažovali a vytváření formulářů, tento nástroj se vám do začátků bude určitě hodit.

ISE pack

Zajímavý modul rozšiřující PowerShell ISE (součást [PowerShell Packu](#)). Do menu přidává několik užitečných rozšíření z nichž některé jsou vidět na následujícím obrázku.



Některé zajímavé zdroje

Samozřejmě, že předchozí seznam není úplný. Pokud vás PowerShell pořádně „chytne“ začnete hledat další dostupné zdroje a zajímavé skripty. Kromě těch několika, které jsem uvedl v některých předcházejících částech by se vám mohly hodit ty následující:

Script Repository

- PoshCode.org
- TechNet Script Center Repository

Podcasty

- [PowerScripting Podcast](#)
- [Get-Scripting Podcast](#)

Twitter

Na Twitteru má účet opravdu hodně lidí z PowerShell světa (včetně například J. Snovera). Můžete se podívat na [PowerShell Twitterers](#) list nebo na Twitteru hledat hashtag #PowerShell.

Samozřejmě je pro vás určitě sledování blogu vývojového týmu PowerShellu:
<http://blogs.msdn.com/powershell/>

PASTI, CHYTÁKY A NECHTĚNÁ PŘEKVAPENÍ

Uživatelé, kteří začínají pracovat s PowerShellem se většinou rychle naučí základní příkazy a postupy. Po čase začnou vytvářet složitější skripty, ve kterých už mohou narazit na záludnosti a chytáky. Pojdme si odhalit některé z nich, abychom se jim už příště vyhnuli a nemuseli dumat nad tím, jestli je to bug, nebo ne.

Automatické odrolování (unrolling/flattening) kolekce

Ve složitějších skriptech budete volat funkce/cmdlety, jejich výsledek uchovávat v proměnné a následně s proměnnou budete pracovat. Potíž může nastat, pokud předpokládáte, že v proměnné máte vždy uloženou kolekci.

Přestavme si následující kód:

```
PS> function GetCollection {
    # vrátí jednoprvkové pole
    @(new-object PsObject -property @{Date=get-date })
}
```

```
PS> $a = GetCollection
```

Jak byste očekávali, že se dostanete na property Date?

"Správně" by to mělo být takto: `$a[0].Date`

Ale PowerShell nám jednoprvkovou kolekci vybalil (unrolled) a do proměnné \$a přiřadil samotný objekt. Na datum se tedy dostaneme takto:

```
PS> $a.Date
```

Stejný problém se netýká jen vámi vytvořených funkcí, ale i vestavěných cmdletů:

```
PS> $a = Get-ChildItem c:\ *windows* # najde adresář s nainstalovanými windows, zřejmě bude jen jeden
```

```
PS> $a.CreationTime # vypíše, kdy byl vytvořen
```

Na této vlastnosti PowerShellu je nepříjemné to, že po volání funkce/cmdletu můžete mít v proměnné uloženou kolekci, nebo vybalený objekt (tj. ne jednoprvkovou kolekci). Ve skriptu pak musíte testovat jestli je proměnná typu pole, nebo ne.

Naštěstí zde existuje jedna velmi jednoduchá technika – zabalení volání funkce/cmdletu do pole:

```
PS> $a = @(GetCollection)
```

```
PS> $a[0].Date
```

8. června 2010 14:14:27

```
PS> $a = @(Get-ChildItem c:\ *windows*)
```

```
PS> $a[0].CreationTime
```

2. listopadu 2006 12:18:34

Na odrolování můžete narazit nejen u polí, ale i u dalších objektů, [DataSety](#) nevyjímaje.

Práce s \$null místo kolekce

Tento případ tak trochu souvisí s předchozím. Předpokládejte tento kód:

```
PS> function GetRandomNumbers {  
    param([int]$min, [int]$max)  
    if ($min -lt $max) {  
        0..10 | % { Get-Random -min $min -max $max }  
    }  
}  
  
PS> $numbers = GetRandomNumbers 0 -10  
  
PS> $numbers | % { write-host Number: $_ }
```

Number:

Co se stane, když zkusíte zadat minimum větší než maximum? Vypíše se pouze "Number: " a žádné číslo. Důvodem je, že z funkce GetRandomNumbers se nám *nic* nevrátilo. Toto *nic* pak bylo při přiřazení do proměnné \$numbers interpretováno jako \$null. A protože je naprosto legální poslat do pipeline \$null, místo čísla se nám vypsalo \$null, jehož reprezentace je prázdný string.

Náprava je opět velmi jednoduchá:

```
PS> $numbers = @(GetRandomNumbers 0 -10)  
  
PS> $numbers | % { write-host Number: $_ }
```

Do proměnné \$numbers se nám tentokrát uložilo prázdné pole, což je přesně to, co jsme chtěli.

Jen podotýkám, že z výrazu @(\$null) nám prázdné pole nevznikne!

Jak si více zamotat hlavu

Pokud máte pocit, že už *víte, jak na to*, možná teprve teď se vám rozsvítí. Opět začneme příkladem.

```
PS> function ReturnNothing { }
```

```
PS> ReturnNothing | % { "some value passed" } # nevypíše nic
```

Tento kus kódu se chová tak, jak byste intuitivně čekali. *Když funkce ReturnNothing nic nevrací, tak se do pipeline nic nepošle.*

```
PS> function ReturnNull { $null }
```

```
PS> ReturnNull | % { "some value passed" } # vypíše řetězec 1x
```

A tady, když se zamyslíme, kód funguje opět podle očekávání. Funkce něco vrací (i když je to jen \$null), tedy do pipeline něco posílá. Proto se řetězec vypíše.

```
PS> $nothing = ReturnNothing
```

```
PS> $rnull = ReturnNull
```

```
PS> $nothing | % { "some value passed" } # vypíše řetězec 1x
```

```
PS> $rnull | % { "some value passed" } # vypíše řetězec 1x
```

Po přiřazení do proměnné, se výsledek *nic* z ReturnNothing transformuje na \$null. Chová se tedy stejně jako u ReturnNull.

A nyní asi možná tušíte, že když obě volání funkcí uzavřete do operátoru @(..), bude se kód chovat opět intuitivně.

```
PS> $nothing = @(ReturnNothing)
```

```
PS> $rnull = @(ReturnNull)
```

```
PS> $nothing | % { "some value passed" } # nevypíše nic
```

```
PS> $rnull | % { "some value passed" } # vypíše řetězec 1x
```

Co to znamená? Pokud explicitně vrátíte \$null, musíte s tím počítat, protože i tento \$null bude v pipeline zpracován.

Pokud byste si o tomto rysu PowerShellu chtěli přečíst ještě něco víc, v článku [Effective PowerShell Item 8: Output Cardinality - Scalars, Collections and Empty Sets - Oh My!](#) jej najdete přehledně popsany.

Automatické vrácení hodnot z funkce

Na začátek opět jedna krátká ukázka:

```
PS> function GetArrayList {
```

```
    $a = new-object Collections.ArrayList
```

```
    $a.Add(10)
```

```
    $a.Add('test')
```

```
    $a
```

```
}
```

```
PS> GetArrayList
```

```
0
```

1

10

test

Víte, co se stalo? Možná byste očekávali, že se vypíše pouze 10 a "test". Jenže i samotná funkce Add něco vrací – index nově přidané položky. A protože jsme výstup z Add nezachytili, dostal se mezi návratové hodnoty.

Korektně bychom tyto položky mohli vyřadit (minimálně) třemi způsoby:

```
$a.Add(10) | out-null
```

```
$null = $a.Add(10)
```

```
[void]$a.Add(10)
```

Update: čtvrtý způsob využívá přesměrování.

```
$a.Add(10) > $null
```

Záleží na vás, který z nich je vám sympatičtější.

PowerShellí funkce se volá jinak než .NETí

Tento problém překvapí spíše jen začátečníky. Ale pořád se s ním dá setkat.

Spočívá ve způsobu, jak se do funkce (a samozřejmě i cmdletu) předávají parametry. V mnoha programovacích jazycích se uzavřou do závorek a oddělí čárkou. Jenže v PowerShellu se pomocí čárky vytváří pole.

```
PS> function Test {
```

```
    param($a1, $a2)
```

```
    write-host A1: $a1
```

```
    write-host A2: $a2
```

```
}
```

```
PS> Test (1, 2)    # špatně
```

```
PS> Test 1, 2      # špatně; chová se stejně jako předchozí volání
```

```
PS> Test 1 2       # správná varianta
```

V prvním a druhém případě jsme vlastně předali hodnoty pouze prvnímu parametru; do proměnné reprezentované druhým parametrem byl přiřazen \$null.

Předání \$null do .NET metody

V případě, že používáme .NET a potřebujeme do metody, která bere string parametr předat \$null, máme smůlu. Na toto téma byl otevřen [bug](#) a snad se ho podaří pořešit do příští verze.

Na onom reportu je k nalezení také workaround. Tj. je to možné, ale ... nepěkné.

Priorita operátorů

Z času na čas vás může zaskočit, proč se váš výraz vyhodnocuje jinak, než jste zamýšleli. V tom případě je možná na vině priorita operátorů.

```
PS> $x = 2
```

```
PS> $y = 3
```

```
PS> $a,$b = $x,$y*5
```

Co byste jako programátoři očekávali v proměnných \$a a \$b? Po mé otázce už to zřejmě nebude 2 a 15. Operátor čárky má totiž větší prioritu než násobení.

[Posholic](#) kdysi experimentálně vytvořil [tabulku s prioritou operátorů](#). Možná mezi nimi najdete i nějaké doposud neznámé kousky.

Operátor -f

S operátorem -f pracujte jen tehdy, když víte, že s ním pracujete správně. Podle všeho byla do jeho implementace zanesen [bug](#).

Ve zkratce řečeno ... operátor špatně vyhodnocuje, pokud mu podstrčíme jako pravý operand pole. Vyzkoušet si to můžete na tomto příkladě:

```
PS> $a = 1,2,3
```

```
PS> "{0} a {1}" -f $a
```

```
1 a 2
```

```
PS> $a.Length
```

```
3
```

```
PS> "{0} a {1}" -f $a
```

Error formatting a string: Index (od nuly) musí být větší nebo roven nule a menší než velikost seznamu argumentů..

At line:1 char:13 ...

Vtip je v tom, že \$a není při prvním formátování obalena do PsObject instance a proto se vyhodnotí stejně, jakobychom zapsali "{0} a {1}" -f \$a[0], \$a[1], \$a[2]. Možná vám to připomíná [splatting](#).

Tím, že přistoupíme k property objektu (!), se tento objekt obalí do PsObject. Při vyhodnocení formátování pak je tento objekt konvertován na string a přiřazen do {0}. Do {1} už není co přiřadit a tedy skončíme s chybou.

Více informací najdete na [StackOverflow](#).

Specifikujte typy u parametrů funkcí

PowerShell je dost chytrý a provádí plno konverzí na pozadí, o kterých ani nevíme, ale ne vždy je schopen si poradit. [Příkladem](#) může být toto:

```
PS> Function Test {
```

```
    param($n, $k)
```

```
if($n -lt 0 -Or $k -lt 0) {
    throw "Negative argument in Test"
}
```

```
"Processing"
}
```

PS> Test 1 -2

Processing

Přirozeně byste očekávali, že funkce vyhodí výjimku. Bohužel se tak nestane. Parametry funkce nemají specifikován typ a tedy PowerShell z nějakých důvodů konvertoval argumenty na *string*.

Upravte hlavičku:

```
PS> Function Test {
    param([int]$n, [int]$k)
    ...
}
```

PS> Test 1 -2

Negative argument in Test

At line:4 char:14 ...

A dostanete očekávaný výsledek.

Generické typy

Vývojáři rychle zjistí, že PowerShell má omezenou podporu generických typů. Můžeme sice vytvořit například `List<string>` (syntaxe ze C#), ale:

- Už nemůžeme volat generickou metodu v negenerické třídě.
- Někdy je potřeba specifikovat fullname typu, viz. [Powershell Generic Collections](#) na StackOverflow.

Díky tomu nemůžeme plnohodnotně prozkoumávat neznámé assembly a musíme si pomáhat berličkami jako [Invoking Generic Methods on Non-Generic Classes in PowerShell](#).

Vyhodnocování proměnných v řetězcích

Čas od času se najde někdo, kdo se potýká s vyhodnocováním proměnných v řetězci:

```
PS> $date = get-date
```

```
PS> "Ted' je $date.Hour hodin" # špatně
```

```
PS> "Ted' je $($date.Hour) hodin" # správně
```

Pozor na proměnnou \$args

Proměnná \$args se používá hlavně u jednoduchých funkcí. Musíme ale vědět, kdy ji použít.

```
PS> function Test {
```

```
    Write-Host args: $args
```

```
    1..3 | Foreach-Object { write-host args je $args }
```

```
}
```

```
PS> Test 1 2 3
```

```
args: 1 2 3
```

```
args je
```

```
args je
```

```
args je
```

Jaktože proměnná nebyla vyhodnocena v rámci cmdletu Foreach-Object?

Důvod je velmi prostý. Proměnná \$args je automatická a inicializuje se v rámci každého volání funkce, skriptu, nebo *scriptblocku*. V tomto případě byl proveden *scriptblock*. V rámci toho byla vytvořena automatická proměnná \$args, navázaná defaultně na \$null.

Způsobů, jak toto obejít je více. Nejjednodušší z nich je odložit si \$args do speciální proměnné a tu pak používat, tj. \$a = \$args ... write-host args je \$a.

Předávání argumentů externím programům

Z PowerShellu můžete přímo spouštět ostatní programy, má to ovšem jedno *Ale*. Někdy je velmi těžké skloubit způsoby, jakým funguje PowerShellí parser a požadavky na syntaxi argumentů pro daný program. Mohou vám chybět uvozovky, může vám dělat problém středník, mezera.

Pokud potřebujete vědět, jak přesně jsou argumenty předány vašemu programu, použijte program EchoArgs z [PSCX](#) modulu. Jak na to?

```
PS> import-Module pscx
```

```
PS> echoargs "argument" second 3rd 'before;after' -like-switch outside"inside""inside2""inside3"
```

```
Arg 0 is <argument>
```

```
Arg 1 is <second>
```

```
Arg 2 is <3rd>
```

```
Arg 3 is <before;after>
```

```
Arg 4 is <-like-switch>
```

```
Arg 5 is <outsideinsideinside2'inside3'>
```

Přesměrování do souboru a kódování

Možná jste zvyklí přesměrovat výstup z nějakého programu do souboru pomocí `>`, případně `>>`. Pak byste měli vědět, že výstup bude uložen do souboru v UNICODE. Je to ekvivalent ... | out-file soubor - encoding unicode.

Obsah tedy můžete přesměrovat ručně pomocí Out-File a při tom specifikovat kódování.

Závěrem

Snažil jsem se vám tu nastínit problémy, které vás mohou potkat při práci s PowerShellem. Pokud nechcete číst článek celý, doporučuji alespoň prozkoumat *Automatické odrolování* (*unrolling/flattening*) kolekce. Je velmi pravděpodobné, že se s ním jednou setkáte.

FUNKCE

Základní konstrukce při práci s funkcemi

Až do minulého dílu jsme se věnovali primárně práci v konzoli. S tou si vystačíme pro většinu administrátorských úkonů. Pokud ale budeme PowerShell využívat pro složitější zprávu, bude se nám časem hodit možnost psaní skriptů. Dnes si řekneme něco o funkcích. Budeme je považovat za základní stavební jednotku skriptu. Funkce můžete psát velmi krátké pro jednostranné použití nebo složitější např. pro zpracování objektů z roury. Takzvané **advanced funkce** nám například umožní jednoduché vložení nápovědy. Začněme tedy nějakou minimalistickou funkcí.

```
PS C:\ function Get-FreeDiskSpace { (gwmi win32_logicaldisk -filter  
'DeviceID="C:")').freespace/1GB }  
PS C:\ Get-FreeDiskSpace  
196.306255340576
```

Nadefinovali jsme si funkci, která nám po zavolání vrátí volné místo na disku C. Za klíčovým slovem **function** uvedeme jméno funkce a poté ve složených závorkách tělo funkce. Zatím nic světoborného, pojďme do funkce přidat nějaké vstupní parametry. Stejně jako v předchzím případě můžete text celé funkce zapsat přímo do konzole nebo použít některý z editorů, např. PowerShell ISE. Editor nám přináší (mimojiné) výhodu zvýrazňování syntaxe a tím i snadnější hledání chyb.

```
function Get-FreeDiskSpace  
{  
  param  
  (  
    $drive = 'C:'  
  )  
  $filter = 'DeviceID="' + $drive + '"'  
  (Get-WmiObject Win32_LogicalDisk -filter $filter).FreeSpace/1GB  
}
```

Poznámka: Čtenáře už by neměla překvapit konstrukce “1GB”.

Přidali jsme klíčové slovo **param** a použili pro parametr jméno **drive**. Nyní můžeme volit, na jakém disku nás volné místo zajímá, přičemž standardní hodnota je disk C.

```
PS C:\> Get-FreeDiskSpace
196.615299224854
PS C:\> Get-FreeDiskSpace C:
196.615299224854
PS C:\> Get-FreeDiskSpace D:
0
PS C:\> Get-FreeDiskSpace -drive C:
196.615299224854
```

V prvním případě voláme funkci bez parametrů a vráceno je množství volného místa na disku C, stejně jako ve druhém případě, kde jsme – trochu zbytečně – uvedli jméno disku. Třetí příklad na mém počítači vrátí nulu (disk D není v systému). Ve čtvrtém případě voláme funkci i se jménem parametru. Toto je vhodné při větším množství parametrů – zlepšujeme čitelnost kódu.

Poznámka: Někteří z vás možná znají z jiných jazyků proměnnou s názvem args. PowerShell jí také disponuje, můžeme ji využít, ale při tvorbě vlastních funkcí bych se ve většině případů klaněl k pojmenování parametrů tak, jako v předchozím případě. Nicméně pouze pro názornost:

```
PS C:\> function Get-Args { "Pocet parametru: $($args.Count)"; "Parametry: $args" }
PS C:\> Get-Args aaa bbb ccc
Pocet parametru: 3
Parametry: aaa bbb ccc
```

Zajímavou možností je při definici parametrů funkce určit rovnou i jejich typ. Následující definice

```
PS C:\> function test { param ([int]$text) }
```

Nám při následujících typech volání buď projdou bez chyby nebo oznámí, že máme problém.

```
PS C:\> test 1
PS C:\> test aaa
test : Cannot process argument transformation on parameter 'text'. Cannot convert value "aaa"
to type "System.Int32".
Error: "Input string was not in a correct format."
At line:1 char:5
+ test <<<< aaa
+ CategoryInfo          : InvalidData: (:) [test], ParameterBindin...mationException
+ FullyQualifiedErrorId : ParameterArgumentTransformationError,test
```

Vidíme, že PowerShell hlásí chybu konverze řetězce „aaa“ na číslo (typ int32). Zajímavým typem proměnné je přepínač (switch), který nám umožňuje rozhodování typu ano-ne. Vraťme se k naší první funkci Get-FreeDiskSpace, nyní přidáme možnost zobrazit množství volného místa v bytech (tak, jak jej vrátí cmdlet Get-WmiObject) nebo přepočtený na GB, jako v naší původní funkci.

```
function Get-FreeDiskSpace
{
```

```

param
(
    $drive = 'C:',
    [switch]$gb
)
$filter = 'DeviceID="' + $drive + '"'
if($gb)
{
    (Get-WmiObject Win32_LogicalDisk -filter $filter).FreeSpace/1GB
}
else
{
    (Get-WmiObject Win32_LogicalDisk -filter $filter).FreeSpace
}
}

```

Parametr **gb** je typu [switch] a jeho uvedení při volání funkce ukáže volné místo v GB.

```

PS C:\> Get-FreeDiskSpace
211123945472
PS C:\> Get-FreeDiskSpace -gb
196.624496459961

```

Poznámka: Všimněte si, že při zadávání jednotlivých parametrů funkce můžete používat pro doplňování jmen klávesu “Tab” stejně jako při práci s cmdlety.

Pokročilé funkce

Pokud znáte funkce z PowerShellu v1, nebyly pro vás předchozí řádky vůbec žádnou novinkou. V PowerShellu v2 byl ovšem uveden concept takzvaných **advanced functions** (osobně nemám moc rád překlady zažitých anglických názvů do češtiny a nadpis této kapitoly je výjimkou).

Advanced functions přidávají několik klíčových slov a umožňují lepší kontrolu parametrů. Další přidanou hodnotou je možnost tvorby velice pěkné nápovědy (stejně jako mají cmdlety). Ukažme si, jak by mohla definice vstupního parametru vypadat.

```

function Show-ParameterAttribute
{
    param (
        [Parameter(
            Mandatory=$true,
            Position=0,
            ParameterSetName="set1",
            ValueFromPipeline=$false,
            ValueFromPipelineByPropertyName=$false,
            ValueFromRemainingArguments=$false,
            HelpMessage="Enter parameter #1.")
        [Alias("p1")]
    )
}

```

```

    [int]
    $param1
)
Write-Output $param1
}

```

Vidíme, že se nám možnosti pro definici parametrů pěkně rozrostly. Význam většiny atributů je zřejmě jasný z názvů, pojďme se nejdříve podívat, jak funguje naše funkce a poté si povíme o attributech podrobněji.

```

PS C:\> Show-ParameterAttribute
cmdlet Show-ParameterAttribute at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
param1: !?
Enter parameter #1.
param1: 0
0
PS C:\> Show-ParameterAttribute -param1 1
1
PS C:\> Show-ParameterAttribute -p1 2
2
PS C:\> Show-ParameterAttribute 3
3
PS C:\> Show-ParameterAttribute 4 5
Show-ParameterAttribute : A positional parameter cannot be found that accepts argument '5'.
At line:1 char:24
+ Show-ParameterAttribute <<<< 4 5
+ CategoryInfo          : InvalidArgument: (:) [Show-ParameterAttribute],
ParameterBindingException
+ FullyQualifiedErrorId : PositionalParameterNotFound,Show-ParameterAttribute

```

Jelikož jsme parametr nastavili jako **Mandatory**, PowerShell nám při volání funkce bez parametrů oznámí, že musíme zadat hodnotu pro param1. Po vyvolání nápovědy pro parametr (označeno žlutě) se zobrazí text, který jsme zadali jako atribut **HelpMessage**. Při druhém volání jsme zadali parametr i se jménem a při třetím využili možnost námi definovaného aliasu. Ve čtvrtém případě zafungoval atribut **Position**. Poslední volání nám oznámí, že zadaná hodnota (5) nemůže být zpracována, protože nám „nezbyl“ parametr, kam bychom ji uložili. Pokud očekáváme, že na vstup se mohou dostat nějaké další parametry, které nechceme ztratit, můžeme změnit atribut **ValueFromRemainingArguments** na **\$true**. Pokračujeme definicí nového parametru.

```

[Parameter(ValueFromRemainingArguments=$true)]$param2
PS C:\> Show-ParameterAttribute 4 5 6 7
param1: 4, param2: 5 6 7

```

Vidíme, že parametr **param1** obsahuje první hodnotu a zbývající tři jsou obsaženy v **param2** (v těle funkce jsme změnili výpis na: Write-Output "param1: \$param1, param2: \$param2").

Nejenom, že můžeme ovlivňovat chování parametrů, můžeme je rovnou na vstupu funkce testovat na určité hodnoty. Tím nám v těle funkce odpadnou podmínky typu if-else a kód bude tím pádem čitelnější. Můžeme použít následující atributy (takzvané Parametr Validation Attributes).

- **AllowNull** – umožňuje, aby parametr obsahoval hodnotu null. Platí i pokud je parametr nastaven jako mandatory.
- **AllowEmptyString** – parametr může být prázdný řetězec.
- **AllowEmptyCollection** – stejné jako předchozí dva, ale pro prázdnou kolekci.
- **ValidateCount** – určuje minimální a maximální počet argumentů.
- **ValidateLength** – určuje délku parametru.
- **ValidatePattern** – specifikuje regulární výraz, kterým můžeme určit vzor parametru.
- **ValidateRange** – minimální a maximální hodnota parametru.
- **ValidateScript** – může obsahovat skript, který validuje parametr. Pokud skript vrátí hodnotu false, PowerShell zahlásí chybu validace.
- **ValidateSet** – parametr může obsahovat pouze uvedené hodnoty.
- **ValidateNotNull** – hodnota parametru neůže být null.
- **ValidateNotNullOrEmpty** – hodnota nemůže být null nebo prázdný řetězec.

Ukažme si krátký příklad.

```
function Test-Parameter
{
    param (
        [Parameter(Mandatory=$True,Position=0)]
        [string]
        [ValidateSet("David")]
        $jmeno,
        [Parameter(Mandatory=$true,Position=1)]
        [int]
        [ValidateRange(33,33)]
        $vek
    )
    Write-Output "Autorem je $jmeno a je mu $vek let."
}
```

Tato funkce vypíše, kdo je autorem tohoto článku, ovšem pouze za podmínky, že zadáme správně jméno a věk.

```
PS C:\> Test-Parameter Petr 33
```

```
Test-Parameter : Cannot validate argument on parameter 'jmeno'. The argument "Petr" does
not belong to the set "David" specified by the ValidateSet attribute. Supply an argument that
is in the set and then try the command again.
```

```
At line:1 char:15
```

```
+ Test-Parameter <<<< Petr 33
```

```
+ CategoryInfo          : InvalidData: (:) [Test-Parameter], ParameterBindingValidationException
```

```
+ FullyQualifiedErrorId : ParameterArgumentValidationError,Test-Parameter
```

```
PS C:\> Test-Parameter David 32
```

Test-Parameter : Cannot validate argument on parameter 'vek'. The 32 argument is less than the minimum allowed range of 33. Supply an argument that is greater than 33 and then try the command again.

At line:1 char:15

+ Test-Parameter <<<< David 32

+ CategoryInfo : InvalidData: (:) [Test-Parameter], ParameterBindingValidationException

+ FullyQualifiedErrorId : ParameterArgumentValidationError,Test-Parameter

PS C:\Scripts> Test-Parameter David 33

Autorem je David a je mu 33 let.

Vidíme, že jsme zkusili zadat špatně buď věk nebo jméno a v obou případech jsme obdrželi chybové hlášení. Až v posledním případě proběhlo vše podle předpokladů.

Tvorba nápovědy

Posledním dnešním tématem bude vytvoření nápovědy pro naši vlastní funkci. Asi jako většina lidí se mohu přiznat k tomu, že jsem líný psát dokumentaci k vlastním skriptům. I když netvrdím, že ve verzi 2 se za vás dokumentace napíše sama, alespoň nám PowerShell tím trošku ulehčil práci. V každém případě jde ale především o naši vlastní vůli zadat těch pár klíčových slov navíc.

Určitě už jste použili v kódu znak # - cokoli od tohoto znaku do konce řádky je bráno jako komentář. Ve verzi 2 přibyla možnost použít znaky <# a #> a cokoli mezi těmito znaky je bráno jako komentář, tento komentář může zasahovat přes více řádek. Této vlastnosti lze využít právě pro vytvoření vlastní nápovědy – jedinou podmínkou je, že použijeme speciální klíčová slova.

Pojďme si vytvořit help pro první funkci z tohoto článku.

function Get-FreeDiskSpace

{

<#

.Synopsis

Funkce zjišťuje volné místo na disku.

.Description

Pomocí WMI třídy Win32_LogicalDrive zjistí volné místo na zadaném disku.

.Parameter drive

Určuje disk, pro který zjišťujeme volné místo.

.Example

Get-FreeDiskSpace

129.7597

.Example

```
Get-FreeDiskSpace -drive D:  
10.86
```

.Link

```
Get-WmiObject
```

```
#>
```

```
param
```

```
(  
    $drive = 'C:'  
)
```

```
$filter = 'DeviceID="' + $drive + '"'  
(Get-WmiObject Win32_LogicalDisk -filter $filter).FreeSpace/1GB  
  
}
```

Při volání nápovědy se zobrazí následující text.

```
PS C:\> Get-Help Get-FreeDiskSpace -Full
```

NAME

Get-FreeDiskSpace

SYNOPSIS

Funkce zjišťuje volné místo na disku.

SYNTAX

```
Get-FreeDiskSpace [[-drive] <Object>] [<CommonParameters>]
```

DESCRIPTION

Pomocí WMI třídy Win32_LogicalDrive zjistí volné místo na zadaném disku.

PARAMETERS

-drive <Object>

Určuje disk, pro který zjišťujeme volné místo.

Required?	false
-----------	-------

Position?	1
-----------	---

Default value	
---------------	--

Accept pipeline input?	false
------------------------	-------

Accept wildcard characters?	
-----------------------------	--

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer and OutVariable. For more information, type, "get-help about_commonparameters".

INPUTS

OUTPUTS

----- EXAMPLE 1 -----

```
C:\PS>Get-FreeDiskSpace  
129.7597
```

----- EXAMPLE 2 -----

```
C:\PS>Get-FreeDiskSpace -drive D:  
10.86
```

RELATED LINKS

[Get-WmiObject](#)

Vidíme, že bez zbytečně velké námahy máme vygenerovanou velice pěknou nápovědu.

Tím bychom pro dnešek skončili. Pokud vás tvorba funkcí zaujala, zkuste se podívat na následující témata nápovědy:

- [about_functions_advanced](#)
- [about_functions_advanced_methods](#)
- [about_functions_advanced_parameters](#)
- [about_Comment_Based_Help](#)
- V minulém díle jsme se plně věnovali funkcím. I když jsem chtěl dnes navázat tématem skripty a moduly, rozhodl jsem se téma funkcí ještě trochu rozšířit použitím atributu `CmdletBinding`. Vzhledem k tomu, že funkce považuji za základní stavební kameny pokročilejšího skriptování, nemyslím, že by to bylo na škodu. O skriptech a modulech si tedy povíme až v některém z příštích pokračování.

• Opravdu pokročilé funkce

- *Použití [`CmdletBinding()`]*
- Možná už jste četli tématickou nápovědu **about_Common_Parameters** a slyšeli jste, že je možné tyto parametry využívat ve vašich funkcích. Malá zmínka je v další tématické nápovědě (**about_Functions_CmdletBindingAttribute**) a tato dale odkazuje na MSDN, kde se můžete dočíst více. Abyste se nemuseli prokousávat celou dokumentací, ukážeme si jednoduché použití atributu **CmdletBinding**.
- Mějme základní “tupou” funkci.
- ```
PS C:\> function secti {param($a,$b) $a+$b}
PS C:\> secti 5 6
11
```
- Zjistíme, jaké má funkce parametry:
- ```
PS C:\> (gcm secti).Parameters | select -expand Keys  
a  
b
```
- Z minulého dílu víme, že můžeme do funkce přidat nápovědu, validaci parametrů, atd., ale pořád nevidíme žádný z “common parameters”. Přidejme tedy atribut **CmdletBinding**.

- PS C:\> function secti {[CmdletBinding()] param(\$a,\$b) \$a+\$b}
PS C:\> secti 1 5
6
- PS C:\> (gcm secti).Parameters | select -expand Keys
a
b
Verbose
Debug
ErrorAction
WarningAction
ErrorVariable
WarningVariable
OutVariable
OutBuffer
- Bez jakékoli větší práce nám tvůrci PowerShellu dodali další zajímavé parametry.
- **Pozor:** Tímto jsme nevytvořili vlastní cmdlet. Stále se jedná o funkci. Cmdlety jsou odvozené od tříd [Cmdlet](#) nebo [PsCmdlet](#). Možná ověřit např. i následujícím způsobem:
- PS C:\> (Get-Command Get-Command).GetType().FullName
System.Management.Automation.CmdletInfo
- PS C:\> (Get-Command secti).CmdletBinding
True
- PS C:\> (Get-Command secti).GetType().FullName
System.Management.Automation.FunctionInfo
- Více k [CmdletInfo](#) a [FunctionInfo](#) lze dohledat opět na MSDN.
- Vytvořme si novou funkci, která bude opět sčítat dvě čísla.
- function Get-Soucet
{
[CmdletBinding()]
param (
[int]\$a = 1,
[int]\$b = 1
)
Write-Verbose "Vstupni cislo a: \$a"
Write-Verbose "Vstupni cislo b: \$b"
Write-Verbose "Pred scitanim."
\$soucet = \$a + \$b
Write-Verbose "Sectenno jako: \$soucet"
Write-Host "Soucet cisel \$a a \$b je \$soucet."
Write-Verbose "Konec funkce."
}
• Povšimněte si použití **Write-Verbose**. Jedná se o naše vlastní “kontrolní” výpisy, kterými můžeme zjišťovat stav funkce v průběhu jejího běhu.
- PS C:\Scripts> Get-Soucet
Soucet cisel 1 a 1 je 2.
PS C:\Scripts> Get-Soucet 5 5
Soucet cisel 5 a 5 je 10.
PS C:\Scripts> Get-Soucet 5 5 -Verbose
VERBOSE: Vstupni cislo a: 5
VERBOSE: Vstupni cislo b: 5

VERBOSE: Před scitaním.

VERBOSE: Secteno jako: 10

Soucet čísel 5 a 5 je 10.

VERBOSE: Konec funkce.

- Vidíte, že při použití parametru **Verbose** jsme dostali navíc informace, které se nám mohou hodit v průběhu ladění. Pokud si tuto techniku zapamatujete, odpadne vám běžný problém se zadáváním **Write-Host** na různá místa a poté jeho odstraňování.
- *Chcete opravdu zmáčknout to červené tlačítko?*
- Pro někoho jsou možná dialogová okna s nápisy – “Are you sure you want to ...” (doplňte dle libosti) – otravná, ale zřejmě pouze do doby, než jsou na konci okna slova “doména” a “smazat” nebezpečně blízko u sebe. Administrace v PowerShellu (nebo spíše obecně v příkazové řádce, skriptu, ...) je skvělá, rychlá, méně náchylná na lidské chyby, ale má samozřejmě svá rizika. Například spojení cmdletů **Get-ADUser** a **Remove-ADUser** na jedné řádce vám může bez dobrého filtrování způsobit docela těžkou hlavu (poděkujme za **Enable-ADOptionalFeature**).
- Možnost vložit so svých skriptů dotaz na provedení akce se hodí při jakékoli destruktivní operaci. PowerShell v2 nám toto umožňuje, pojďme se podívat jak na to. Nejdříve si ukážeme celou funkci a poté vysvětlíme jak funguje.
- ```
function Invoke-Phobia
{
 [CmdletBinding(SupportsShouldProcess=$true)]
 param (
 [int]$cislo,
 [switch]$mocnina
)
 "Soucet: $($cislo+$cislo)"
 if ($mocnina)
 {
 if ($PsCmdlet.ShouldProcess("$cislo", "Pata mocnina"))
 {
 "Mocnina $([Math]::Pow($cislo, 5))"
 }
 }
}
```
- Jak vidíte, přidali jsme parametr **SupportsShouldProcess**. Funkce nám okamžitě nabízí dva nové parametry – **WhatIf** a **Confirm**.
- ```
PS C:> (gcm Invoke-Phobia).Parameters | select -expand Keys
cislo
mocnina
Verbose
Debug
ErrorAction
WarningAction
ErrorVariable
WarningVariable
OutVariable
OutBuffer
WhatIf
Confirm
```
- Tyto parametry lze použít u velké části cmdletů, takže například

- PS C:\> Remove-Item *.ps1 -WhatIf
What if: Performing operation "Remove File" on Target "C:\Scripts\profile.ps1".
What if: Performing operation "Remove File" on Target "C:\Scripts\VeryVERYuseful-DO_NOT_DELETE.ps1".
- nám dá šanci zareagovat při nesprávném použití. A teď zpět k naší funkci – provádí dvě operace, jednu nedestruktivní (součet – provádí se při každém volání) a jednu destruktivní (pátou mocninu – provede se pouze, pokud uvedeme switch mocnina). Destruktivní proto, že náš klient má fobii na vysoká čísla a mocnina by ho mohla rozrušit. Pojďme si tedy naši funkci vyzkoušet.
- PS C:\> Invoke-Phobia 5
Soucet: 10
- PS C:\> Invoke-Phobia 5 -WhatIf
Soucet: 10
- PS C:\> Invoke-Phobia 5 -mocnina
Soucet: 10
Mocnina 3125
- PS C:\> Invoke-Phobia 5 -mocnina -WhatIf
Soucet: 10
What if: Performing operation "Pata mocnina" on Target "5".
- PS C:\> Invoke-Phobia 5 -mocnina -Confirm
Soucet: 10
- Confirm
Are you sure you want to perform this action?
Performing operation "Pata mocnina" on Target "5".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
Mocnina 3125
- Při prvním volání dostaneme pouhý součet hodnoty. Stejně tak se nic převratného nestane, pokud použijeme parametr **WhatIf** bez volání mocniny. Zajímavá je pro nás až třetí varianta – volání mocniny a zároveň uvedení **WhatIf**. PowerShell nám ukáže, co by se mohlo stát v případě, že funkci spustíme bez **WhatIf**. Všimněte si, že parametry, které jsme uváděli u metody [ShouldProcess](#) se nám objevily ve výpisu. Použití **Confirm** je poté již jen rutinní záležitostí.
- Použití WhatIf při každém volání
- Představte si situaci, kdy pracujete se skutečně kritickým systémem a chcete mít jistotu, že v předchozích odstavcích zmiňované „bezpečnostní mechanismy“ jsou spouštěné při každém volání „destruktivního“ cmdletu nebo funkce.
- C:\Script>ls | select name | fw -c 5
file1.txt file10.txt file2.txt file3.txt file4.txt
file5.txt file6.txt file7.txt file8.txt file9.txt
- C:\Scripts>Remove-Item * -WhatIf
What if: Performing operation "Remove File" on Target "C:\Scripts\file1.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file10.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file2.txt".
(...)
- C:\Scripts>Remove-Item *
What if: Performing operation "Remove File" on Target "C:\Scripts\file1.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file10.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file2.txt".
(...)

- C:\Scripts>Remove-Item * -Force
What if: Performing operation "Remove File" on Target "C:\Scripts\file1.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file10.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file2.txt".
(...)
- C:\Scripts>ls | select name | fw -c 5
file1.txt file10.txt file2.txt file3.txt file4.txt
file5.txt file6.txt file7.txt file8.txt file9.txt
- Jak vidíte ani parametrem **Force** se nám nepodařilo donutit cmdlet **Remove-Item**, aby smazal dané soubory. Což by se nám mohlo hodit, pokud bychom se pohybovali např. o adresář výš, než jsme nyní. V PowerShellu existuje proměnná, která se jmenuje **WhatIfPreference** a určuje chování parametru **WhatIf**.
- C:\Scripts>ls Variable:\WhatIfPreference | fl *
PSPath : Microsoft.PowerShell.Core\Variable::WhatIfPreference
PSDrive : Variable
PSProvider : Microsoft.PowerShell.Core\Variable
PSIsContainer : False
Name : WhatIfPreference
Description : If true, WhatIf is considered to be enabled for all commands.
Value : False
Visibility : Public
Module :
ModuleName :
Options : None
Attributes : {}
- Z parametru **Description** můžeme vyčíst, jak tato proměnná funguje. Standardně je nastavena na False a ve všech předchozích případech jsem ji ručně nastavil na True.
- C:\Scripts>Remove-Item *
What if: Performing operation "Remove File" on Target "C:\Scripts\file1.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file10.txt".
What if: Performing operation "Remove File" on Target "C:\Scripts\file2.txt".
(...)
- C:\Scripts>\$WhatIfPreference
True
- C:\Scripts>\$WhatIfPreference = \$false
C:\Scripts>Remove-Item *
C:\Scripts>ls | select name | fw -c 5
- **Poznámka:** Pokud chcete některému z kolegů trošku ztížit práci v PowerShellu, myslím že vložení řádky \$WhatIfPreference = \$true do jeho profilu může být docela legrace. Jestli nechte TechNet Flash, tak se možná na chvíli zapotí J
- Samozřejmě by nedávalo smysl, abyste před každým voláním **Remove-Item** (a podobných) vraceli hodnotu proměnné **WhatIfPreference** zpátky na false. Pomocí následujícího zápisu změníte hodnotu jednorázově.
- C:\Scripts>\$WhatIfPreference
True
- C:\Scripts>Remove-Item * -WhatIf:\$False
C:\Scripts>ls
C:\Scripts>

- Tímto bychom ukončili naši miniexkurzi do tajů funkcí. Pokud vás zaujalo použití proměnné **WhatIfPreference** doporučuji k přečtení tématickou nápovědu **about_preference_variables**.

SKRIPTY A MODULY

Další možností, jak uchovat naše PowerShellí výtvořky pro budoucí generace jsou mimo funkcí (probraných v posledních dvou dílech) také skripty a moduly. Dnes si o nich něco povíme.

Skripty

Pokud máte vytvořeno více funkcí, můžete (a zřejmě i budete) je sdružovat do skriptů. Skript může být opět záležitostí na jednu řádku, ale pravděpodobněji budete vytvářet skripty o něco složitější. Skript je v podstatě kus kódu uložený do souboru s příponou PS1. Již v [prvním díle](#) tohoto seriálu jsme si řekli základní informace o spouštění funkcí, proto je již nebudeme opakovat.

```
C:\Scripts> 'Get-Process | Sort-Object Name | Select-Object -First 5' >
skript1.ps1
C:\Scripts> .\skript1.ps1
```

	Handles	NPM (K)	PM (K)	WS (K)	VM (M)	CPU (s)	Id
ProcessName							
42	AESTFltr	3	1224	3804	33	0,14	2240
39	ApMsgFwd	2	964	2480	20	0,08	4056
41	ApntEx	2	948	2932	29	0,06	1680
88	Apoint	3	2028	5780	37	1,02	576
161	BbDevMgr	4	3480	5056	39	209,06	2156

Právě jsme vytvořili první skript, prostým zkopírováním příkazů do textového souboru. Tento soubor jsme spustili a dostali jsme naprosto stejné výsledky jako bez použití skriptu. Až potud žádné překvapení, pojďme dále.

```
C:\Scripts>$processCount = (Get-Process | Measure-Object).Count
C:\Scripts>$processCount
88

C:\Scripts> @'
>> $pc = (Get-Process | Measure-Object).Count
>> Write-Host $pc
>> '@ > script2.ps1

C:\Scripts>.\script2.ps1
88

C:\Scripts> $pc
C:\Scripts>
```

Vidíme, že proměnná **pc** neobsahuje žádnou hodnotu. Přesto skript proběhl, protože **Write-Host** nám počet procesů vypsál. Zde vstupuje do hry oblast platnosti proměnných (anglicky *scope*). Stručně řečeno, proměnné můžeme vytvářet tak, že budou „viditelné“ (uvidíme jejich hodnoty) pouze ve funkci, skriptu nebo globálně. Pokud neuvedeme výslovně oblast, platí všechny proměnné, vytvořené ve skriptu, pouze uvnitř tohoto skriptu a „zvenku“ (konzole) je nemůžeme přečíst. Což byl přesně předchozí případ. Proměnná **pc** vznikla uvnitř skriptu, cmdlet **Write-Host** ji vypsál na konzoli

(uvnitř skriptu) a po jeho skončení zanikla i tato proměnná. Máme několik možností, jak tuto vlastnost obejít.

Uvedení platnosti proměnné v jejím názvu

Nejdříve opět předvedeme a poté si techniku okomentujeme.

```
C:\Scripts> @'
>> $Global:pc = (Get-Process | Measure-Object).Count
>> Write-Host $Global:pc
>> '@ > script3.ps1

C:\Scripts> .\script3.ps1
88

C:\Scripts> $pc
88
```

Skript je stejný, ale před vlastním jménem proměnné jsme uvedli modifikátor **Global**. Tím určujeme, že proměnná bude platit globálně a tedy i v konzoli.

Dot-sourcing

Druhou možností je takzvaný dot-sourcing (nebudu se pouštět do českého překladu Pomocí tečky (.) určíme, že skript má „proběhnout globálně“ Tím zajistíme, že všechny proměnné zůstanou viditelné globálně i po skončení vlastního skriptu.

Nejprve smažeme proměnnou **pc**, která už globálně existuje.

```
C:\Scripts> Remove-Variable pc
C:\Scripts> $pc
C:\Scripts>
```

A nyní spustíme znovu druhý skript, který nám dříve „nefungoval“.

```
C:\Scripts> . .\script2.ps1
89

C:\Scripts> $pc
89
```

Všimněte si tečky uvedené před voláním skriptu. Ta nám zajistí dot-sourcing.

Pokud se ptáte, jestli je možné u funkcí použít parametry, odpověď je ano. Pro parametry a nápovědu platí stejná pravidla jako pro funkce.

Profil

Pokud budete pracovat v PowerShellu delší dobu, začnete si možná upravovat prostředí tak, aby vám více vyhovovalo. Ať již se jedná o vytváření vlastních aliasů, definici proměnných nebo například o vlastní skripty, může se hodit, mít je v konzoli okamžitě připravené k použití. Uvedme si příklad. Často pracujete s měnami a hodilo by se vám, kdybyste měli funkci na převod mezi dolary a korunami jednoduše dostupnou.

```
C:\Scripts> @'
>> function Get-Dolar
>> {
>> param([int]$castka)
>> Write-Output $($castka/18)
>> }
>> '@ > get-dolar.ps1
```

Skript spustíme pomocí dot-sourcingu (jinak by se nám funkce „neobjevila“ s globální platností a můžeme převádět koruny na dolary (prosím teď neřešme funkci jako takovou – jistě je na ní mnoho co vylepšovat).

```
C:\Scripts > . .\get-dolar.ps1
C:\Scripts > Get-Dolar 72
4
```

Vidíme, že funkce funguje a můžeme ji používat. Při dalším spuštění PowerShellu budeme ovšem muset opět zavolat skript, abychom mohli funkci použít. V tomto případě přichází na řadu *profil*. Stručně řečeno je profil skript, který se spouští při startu PowerShellu. Na vašem počítači je uložen v:

```
C:\Scripts> $PROFILE.AllUsersAllHosts
C:\WINDOWS\system32\WindowsPowerShell\v1.0\profile.ps1
```

Standardně není vytvořený a pokud jej nemáte stačí jej vyrobit například v notepadu. V okamžiku, kdy máte profil vytvořený, můžete do něj dávat např. funkce, které často využíváte. Pojďme si tedy takový profil vytvořit.

```
C:\Scripts> gc $PROFILE.AllUsersAllHosts
Get-Content : Cannot find path
'C:\WINDOWS\system32\WindowsPowerShell\v1.0\profile.ps1' because it
does not exist.
At line:1 char:3
+ gc <<<< $PROFILE.AllUsersAllHosts
+ CategoryInfo          : ObjectNotFound:
  (C:\WINDOWS\syst...1.0\profile.ps1:String) [Get-Content],
  ItemNotFoundException
+ FullyQualifiedErrorId :
  PathNotFound,Microsoft.PowerShell.Commands.GetContentCommand
```

```
C:\Scripts> @'
>> function Get-Dolar
>> {
>> param([int]$castka)
>> Write-Output $($castka/18)
>> }
>> '@ > $PROFILE.AllUsersAllHosts
>>
```

```
C:\Scripts> gc $PROFILE.AllUsersAllHosts
function Get-Dolar
{
param([int]$castka)
Write-Output $($castka/18)
}
```

Potřebnou funkci máme uloženou. Nyní je potřeba znovu spustit PowerShell. Po spuštění můžeme vyzkoušet naši známou konverzi.

```
C:\Scripts> Get-Dolar 72
4
```

Více o profilech se můžete dočíst v tématické nápovědě **about_Profiles**. Důležité je například to, že profil není pouze jeden nebo že profily jsou uloženy v různých cestách.

Moduly

Po funkcích a skriptem postoupíme do dalšího levelu a podíváme se na moduly. Zde si dovolím malou odbočku. Několik let zpátky jsem se staral o systém jehož výstupem bylo velké množství různých log souborů. Vzhledem k tomu, že LogParser byl v té době hudbou budoucnosti, hledal jsem možnosti,

jak tyto logy prohledávat. Nakonec jsem skončil u jazyka Perl. Vzhledem k tomu, že se mé skripty rozrůstaly a komplexnost se zvětšovala, potřeboval jsem stále dokonalejší techniky. A tehdy přišel na řadu [CPAN](#) – zdroj stovek skriptů od různých autorů pokrývajících snad všechny myslitelné oblasti. Stačilo jen najít správný skript (modul) a „doinstalovat“ jej do mého prostředí. CPAN dává jako příklad i Bruce Payette ve své knize PowerShell in Action.

Idea modulů je tedy jednoduchá – umožnit různým autorům vytvářet „skripty“ které budou snadno přenositelné mezi počítači a pro složitější úkony budou poskytovat větší funkcionalitu než skripty. V PowerShellu v1 bylo jediným možným rošířením použití tzv. snap-inů. Například vynikající cmdlety pro správu Active Directory od firmy Quest byly poskytovány ve formě snap-inů (tedy v podobě DLL souborů). V současné době je velké množství rozšíření PowerShellu poskytováno ve formě modulů (tedy text). Moduly ovšem mohou být i binární, více viz dále.

Nejdříve si ukážeme práci s již existujícími moduly a poté si ukážeme, jak vytvořit moduly vlastní.

Kontrolní otázka: Zkuste si tipnout – jaký cmdlet použít pro zjištění dostupných modulů? Pokud jste odpověděli **Get-Module**, máte samozřejmě pravdu. Pojdme si ukázat příkazy, které použije při práci s moduly. Opět nejdřív ukážeme celý kód a poté jej okomentujeme.

```
C:\Scripts> Get-Module
C:\Scripts> Get-Module -ListAvailable
```

```
ModuleType Name ExportedCommands
```

```
-----
```

```
Script Cookbook {}
Script CP2010 {}
Script DotNet {}
Manifest FileSystem {}
Script ISEDemo {}
Manifest IsePack {}
Manifest PowerShellPack {}
Manifest PSCodeGen {}
Manifest Pscx {}
Manifest PSImageTools {}
Manifest PSRemoteRegistry {}
Manifest PSRSS {}
Manifest PSSystemTools {}
Manifest PSUserTools {}
Manifest SMS2003 {}
Manifest TaskScheduler {}
Manifest WPK {}
Manifest BitsTransfer {}
```

```
C:\Scripts> Import-Module PSRemoteRegistry
C:\Scripts> gmo
```

```
ModuleType Name ExportedCommands
```

```
-----
```

```
Script PSRemoteRegistry {Get-RegKey, Get-RegBinary, Get-RegQWord, Get-RegMultiString...}
```

```
C:\Scripts> ipmo BitsTransfer
C:\Scripts> ipmo wpk
C:\Scripts> gmo
```

```
ModuleType Name ExportedCommands
```

```
-----
```

```
Script PSRemoteRegistry {Get-RegKey, Get-RegBinary, Get-RegQWord, Get-RegMultiString...}
Manifest bitstransfer {Start-BitsTransfer, Remove-BitsTransfer, Resume-BitsTransfer, Get-BitsT...}
```

```
Script wpk {Get-DependencyProperty, New-ModelVisual3D, New-
DiscreteVector3DKeyFrame...
C:\Scripts> Remove-Module ps*
C:\Scripts> rmo b*
C:\Scripts> New-Label "$((gmo).name)" -show
```

```
C:\Scripts> gmo|rmo
C:\Scripts> gmo
```

Jak jsme již řekli, pro zobrazení *instalovaných* modulů slouží cmdlet **Get-Module** zadaný bez parametrů. Jelikož nebyl žádný modul použit, nezobrazilo se ve výpisu nic. Proto jsme použili parametr **ListAvailable**. Při jeho použití se PowerShell podívá do všech adresářů s moduly (adresáře jsou uvedeny v proměnné prostředí **\$env:PSModulePath**) a vypíše moduly, které můžeme importovat.

Abychom moduly mohli použít, musíme je importovat pomocí Import-Module. Poté je můžeme použít, respektive můžeme volat funkce (nebo cmdlety) v nich obsažené. Zde ukázáno na příkladu modulu WPK (součást Windows PowerShell Pack – více viz [9. část tohoto seriálu](#)). Na konci práce můžeme modul odebrat pomocí **Remove-Module**. Další příkazy pro práci s moduly (včetně aliasů) můžete zobrazit například takto:

```
C:\Scripts> gcm *module* -C cmdlet | sort name |% {"{0} {1}" -f
$_.Name, $(gal -def $_.Name -ea 0)}
Get-Module gmo
Import-Module ipmo
New-Module nmo
New-ModuleManifest
Remove-Module rmo
Test-ModuleManifest
```

A jako vždy – podrobnější informace o práci s moduly najdete v tématické nápovědě **about_modules**.

TIPY A TRIKY – MAPY

Programátorům a administrátorům, kteří objevili možnosti .NETu, PowerShell nabízí možnosti, jak rychle prozkoumat neznámé API. Dnes si ukážeme, jakým způsobem bychom mohli postupovat v případě, že bychom si chtěli osahat kontrol na zobrazování map, [GMap.NET - Great Maps for Windows Forms & Presentation](#). Pro jednoduchost si vybereme kontrol pro Windows Forms. Výsledkem pak bude malá aplikace, která nám vrátí GPS souřadnice vybraného místa.

Poznámka: Příklad bude běžet v pořádku pouze pokud bude PowerShell spuštěn v režimu STA. Při spuštění PowerShellu tedy nezapomeňte na přepínač **-sta**.

V průběhu se seznámíme s některými tipy na práci s enumy, *[out]* parametry a jak zapisovat handlersy k událostem.

Kostra

Budeme potřebovat **Form** objekt, který bude mapový kontrol obsahovat:

```
Add-Type -AssemblyName System.Windows.Forms
$f = New-Object System.Windows.Forms.Form
$f.Size = New-Object System.Drawing.Size 700,500
... místo pro další kód
```

```
[void]$f.showdialog()
```

Programátoři zřejmě nic nového neobjevili, tento kód se opakuje pokaždé, když chceme psát WinForms aplikaci.

Dále si potřebujeme stáhnout příslušné assembly a načíst je. V našem případě půjde o tři assembly, které najdete v příloze. Samozřejmě si je můžete stáhnout i z výše uvedené [adresy](#).

```
# assembly máme uložené v adresáři lib, načteme je všechny takto:
```

```
gci g:\lib\ *.dll | % { Add-Type -path $_.FullName }
```

Závislosti mezi assemblies

Pokud byste potřebovali najít "ty správné" assemblies, které máte načíst, čtěte dál. Jinak můžete pokračovat dál k vytváření kontrolů.

Assembly, kterou musíte načíst, najdete podle projektu. My budeme používat **GMapControl**. Rychle proletíme adresářovou strukturu (autor používá zásadu *co třída, to nový soubor*) a zjistíme, že kontrol se nachází v projektu *GMap.NET.WindowsForms*. Poté vyhledáme **.csproj* soubor a v něm element *AssemblyName*. Tak jsme zjistili, že kontrol se kompiluje do assembly *GMap.NET.WindowsForms.dll*.

Našli jsme první assembly, kterou určitě načíst musíme. Stejně tak ale budeme pravděpodobně potřebovat i další assembly, na kterých tato závisí. Na závislosti se můžeme podívat přes [Reflector](#), ale stejně dobrou práci zvládne i PowerShell.

```
PS> $assembly = [reflection.assembly]::LoadFile('g:\lib\GMap.NET.WindowsForms.dll' )
```

```
PS> $assembly | fl
```

```
CodeBase           : file:///g:\lib\GMap.NET.WindowsForms.dll
EntryPoint         :
EscapedCodeBase    : file:///g:\lib\GMap.NET.WindowsForms.dll
FullName           : GMap.NET.WindowsForms, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=b85b9027b614afef
GlobalAssemblyCache : False
HostContext        : 0
ImageFileMachine   :
ImageRuntimeVersion : v2.0.50727
Location           : g:\lib\GMap.NET.WindowsForms.dll
ManifestModule     : GMap.NET.WindowsForms.dll
MetadataToken      :
PortableExecutableKind :
ReflectionOnly      : False
```

Properties assembly jsme sice ještě nezjistili, ale zajímavá může být určité informace o *FullName*. Ta se vyplatí v případě, kdy nám runtime hlásí, že nemůže načíst nějakou assembly. Je pravděpodobné, že se mu snažíme podstrčit špatnou verzi. Všimněte si také *ImageRuntimeVersion* – ta nám říká, že assembly byla překompilována pro runtime 2.0. Pro aplikace pod .NET 4.0 pak budete potřebovat zřejmě jinou verzi assembly. PowerShell ovšem stále běží na .NET 2.0/3.5, tedy pro naše potřeby máme verzi správnou.

Vraťme se zpátky. Když předhodíme `$assembly` commandletu `Get-Member -membertype method`, zjistíme metody, které nám nabízí. Pro stručnost už je zde nebudu uvádět. Ta správná metoda, která nám zjistí, na kterých ostatních assemblies ta naše závisí, se skrývá pod jménem ...

```
PS> $assembly.GetReferencedAssemblies()
```

Version	Name
-----	----
2.0.0.0	System.Windows.Forms
2.0.0.0	mscorlib
1.5.3.3	GMap.NET.Core
2.0.0.0	System
2.0.0.0	System.Drawing

Víme tedy, že assembly *GMap.NET.Core* budeme potřebovat načíst také. A když se podíváme na ni ...

```
PS> $assembly2 = [reflection.assembly]::LoadFile('g:\lib\GMap.NET.Core.dll')
```

```
PS> $assembly2.GetReferencedAssemblies()
```

Version	Name
-----	----
...	
1.0.66.0	System.Data.SQLite

Zjistili jsme závislost na *System.Data.SQLite*. Z tohoto důvodu jsou všechny tyto tři assembly přibaleny k tomuto článku a výše je načítáme pomocí příkazu:

```
gci .. | % { Add-Type .. }
```

Vytvoření kontrolky

Na oficiálních stránkách bohužel mnoho dokumentace není. Inspirovat se můžeme převážně jen z demo aplikace, která je mimochodem dostupná pro Windows Forms i WPF. Zkopírujeme si základní kód na inicializaci kontrolu:

```
$map = New-Object GMap.NET.WindowsForms.GMapControl
$map.Anchor = 'top', 'bottom', 'left', 'right'
$map.Location = New-Object System.Drawing.Point 0,0
$map.MapType = 'GoogleMap'
$map.MarkersEnabled = $true
$map.MaxZoom = 17
$map.MinZoom = 2
$map.MouseWheelZoomType = 'MousePositionAndCenter'
$map.ShowTileGridLines = $false
$map.Size = new-object System.Drawing.Size 700, 500
```

```
$map.Zoom = 16  
$map.Position = new-object GMap.NET.PointLatLng 50.207, 16.849
```

Povšimněte si properties `Anchor`, `MapType` a `MouseWheelZoomType`. PowerShell nám velmi pomáhá, při práci s enumy. Pokud se snažíme přiřadit do proměnné typu *enum* hodnotu typu *string*, PowerShell ji automaticky zkonvertuje na příslušný enum. Když přiřazujeme pole stringů a daný enum má přiřazený atribut `[FlagsAttribute]`, pak všechny stringové hodnoty jsou zkonvertovány na enumové hodnoty a následně kombinovány dohromady. Můžete si to ověřit například takto:
`$map.Anchor -band [system.windows.forms.anchorstyles]::right`.

Abychom si mohli přepínat mezi více typy map, přidáme si ještě na formulář combo box. Ten bude reagovat na změnu indexu a nastaví příslušný typ mapy do objektu `$map`:

```
$change = new-object System.Windows.Forms.ComboBox  
$change.Anchor = 'top', 'right'  
$change.Location = New-Object System.Drawing.Point 615,0  
$change.DropDownStyle = 'DropDownList';  
$change.FormattingEnabled = $true;  
$change.Items.AddRange(@( 'GoogleMap', 'GoogleSatellite', 'GoogleTerrain', 'GoogleHybrid' ))  
$change.Size = new-object System.Drawing.Size(80, 21);  
$change.TabIndex = 2;  
$change.add_SelectedValueChanged({ $map.MapType = $change.SelectedItem })  
$change.SelectedIndex = 1
```

Hodnoty *GoogleMap* až *GoogleHybrid* jsou opět hodnoty enumu. Jak zjistíme dostupné hodnoty? Například takto:

```
PS> $map.MapType.gettype().fullname  
GMap.NET.MapType  
PS> [enum]::getnames([GMap.NET.MapType])  
None  
GoogleMap  
GoogleSatellite  
GoogleLabels  
GoogleTerrain  
GoogleHybrid  
GoogleMapChina  
GoogleSatelliteChina  
GoogleLabelsChina  
GoogleTerrainChina  
GoogleHybridChina
```

....

K hodnotám enumu se přistupuje – pokud nemůžeme zrovna použít automatické konverze ze stringu – jako k statickým properties. Tedy jméno enumu uzavřeme do hranatých závorek a k hodnotám se dostaneme přes dvojtečkovou notaci, viz. příklad s `AnchorStyles`:
`[system.windows.forms.anchorstyles]::right`

Možná jste si všimli, jakým způsobem definujeme handler pro událost `SelectedValueChanged` – voláme metodu se jménem `add_jmeno-udalosti` a jako parametr předáváme *scriptblock*, který se má vykonat. Handlers mívají dva parametry, které se často ale ani nevyužijí, jako v tomto případě. Pokud byste přesto potřebovali jeden z nich přečíst, níže je zobrazen handler na událost *Click*, kde se získává aktuální pozice myši.

Můžeme si vyzkoušet, jestli bylo naše dosavadní snažení k něčemu dobré. Kontroly vložíme do kolekce a zobrazíme formulář.

```
$f.Controls.Add($change)
$f.Controls.Add($map)
[void]$f.ShowDialog()
```

Naši malou aplikaci teď můžeme spustit. Rolováním kolečka myši měníme zoom. Klikem pravým tlačítkem a tažením pak posouváme výřez.

Vidíme, že vcelku zadarmo jsme získali aplikaci, která je schopna bez prohlížeče zobrazovat mapy a dokonce je cachovat. Vnitřně totiž používá SQLite, kde si uchovává stažené náhledy a funguje tak i v případě, že zrovna není dostupné síťové připojení.

Přidání křížku

Pokud chceme ještě přidat na mapu značku, kterou bychom zacílili na námi vybraný bod, potřebujeme přidat vrstvu a do ní značku zasadit:

```
$overlay = New-Object GMap.NET.WindowsForms.GMapOverlay $map, "point"
$map.Overlays.Add($overlay)
$marker = New-Object GMap.NET.WindowsForms.Markers.GMapMarkerCross($map.Position)
$overlay.Markers.Add($marker)
```

Na mapě se nám tak bude zobrazovat červený křížek. Při kliku chceme, aby se křížek přesunul na místo, kam jsme kliknuli. To docílíme opět zavěšením na události `Click`.

```
$map.add_Click({
    param($s, $e)
    $marker.Position = $map.FromLocalToLatLng($e.X, $e.y)
    $map.Tag = $marker.Position.Lat,$marker.Position.Lng
})
```

Všimněte si, že tentokrát jsme použili *scriptblock* s parametry. Druhý parametr obsahuje umístění myši v okamžiku kliknutí. Podle nich nastavíme červenému křížku nové souřadnice a uložíme je do property `$map.Tag`.

Poté, co zavěříme formulář, budeme mít v property `Tag` poslední kliknuté souřadnice. Přečteme je tedy a vrátíme jako výsledek skriptu. Poslední řádek skriptu tedy bude:

```
$map.Tag
```

Nastavení počátečních souřadnic

Ve skriptu máme nyní počáteční souřadnice nastavené napevno. My ovšem máme i možnost si souřadnice vyhledat. Níže uvedený kód vložte kdekoli za inicializaci mapového kontrolu a před zobrazení formuláře:

```
[GMap.NET.GeoCoderStatusCode]$status = 'G_GEO_SUCCESS'

$loc = [GMap.NET.GMaps]::Instance.GetLatLngFromGeocoder("Kralický sněžník", [ref]
$status);

if ($status -eq 'G_GEO_SUCCESS') {
    Write-Host "Nalezen Kralický Sněžník, souřadnice: $($loc.Lat), $($loc.Lng)"
    $lat,$lng = $loc.lat,$loc.Lng
    $map.Position = new-object GMap.NET.PointLatLng $lat, $lng
} else {
    $lat,$lng = 49.6034664,17.254005
}
```

Zde si povšimněte použití metody `GetLatLngFromGeocoder`. Druhý parametr by měl být předáván jako `[out]`. Jenže PowerShell nic takového nezná. V PowerShellu se oba případy (`[ref]` i `[out]`) předávají jako `[ref]`. Příklad můžete vidět například na <http://wiki.poshcode.org>.

Někdy se nám může hodit, když známe, jak zapsat typovou proměnnou:

`[GMap.NET.GeoCoderStatusCode]$status`. Takovým způsobem určíme typ proměnné `$status`, který zaručuje, že do proměnné vždy můžeme uložit pouze hodnotu typu `[GMap.NET.GeoCoderStatusCode]`. Opět platí, že pokud se snažíme do ní uložit stringovou hodnotu, je tato hodnota konvertována a v případě úspěchu uložena. Můžeme si to ukázat na tomto příkladu:

```
[GMap.NET.GeoCoderStatusCode]$status = 'G_GEO_SUCCESS'

$status = 'test' #chyba

    Cannot convert value "test" to type "GMap.NET.GeoCoderStatusCode" due to invalid
enumeration values.....

$status = 'G_GEO_TOO_MANY_QUERIES' #úspěšně
```

Povedlo se?

S trochou štěstí jste se dostali až k opravdu funkčnímu kódu a dozvěděli se pár drobných triků, které vám mohou pomoci programovat téměř jako v C#. Opravdu jen téměř. Nezapomínejme totiž, že PowerShell je primárně skriptovací jazyk, který rozhodně nemá ambice stát se plnohodnotným objektově orientovaným jazykem.

Download

- Skript: [maps-pub.ps1](#)
- Assembly: [lib.zip](#)

MODULY

Při našem [posledním setkání](#) jsme si ukázali úvod do modulů. Naučili jsme se, jak je importovat do PowerShellu a jak zjistit, které moduly máme již importované. Zajisté jste dali na mou radu a přečetli si tématickou nápovědu **about_modules**. J Dnes si ukážeme, jak vytvořit modul vlastní.

Nejdříve si „hodíme“ potřebné příkazy do konzole:

```
PS C:\> $wc = New-Object Net.WebClient
PS C:\> $link =
'http://www.cnb.cz/cs/financni_trhy/devizovy_trh/kurzy_devizoveho_trhu/
denni_kurz.txt'
PS C:\> $wc.DownloadFile($link, 'C:\Scripts\kurzy\kurzy.txt')
PS C:\> gc 'C:\Scripts\kurzy\kurzy.txt' |? { $_ -match 'EUR' } |% { $_
-match '\|(?<kurz>\d{2}.\d{1,3})$' | Out-Null;
[double]$matches.kurz.replace(',','.')
25,01
```

Na první řádce vytvoříme objekt [WebClient](#). Pomocí tohoto objektu můžeme stahovat data z dané webové stránky (velmi, velmi zjednodušeně řečeno – tato třída toho umí daleko více, ale pro nás je momentálně důležité, že máme proměnnou, pomocí které můžeme stáhnout text dané stránky). Do proměnné **\$link** si uložíme cestu, ze které budeme stahovat daný soubor (podle cesty je zřejmé, že nám půjde o kurzy měn na stránkách ČNB). Na třetím řádku použijeme metodu [DownloadFile](#) a uložíme daný soubor na lokální disk. Na posledním řádku už pouze filtrujeme ze souboru potřebné údaje – pro filtrování jsme použili regulární výraz (o regulárních výrazech si povíme v některém z dalších pokračování). Na posledním řádku už je vidět kurz pro aktuální den.

Je vidět, že jsme obdrželi požadovaná data, ale k dokonalosti nám samozřejmě hodně chybí. Pokud budeme chtít spustit stejné příkazy zítra, nebude příliš efektivní přepisovat celý kód znovu. Proto by se nám hodilo uložit všechny potřebné příkazy do jednoho souboru. Mohli bychom použít skript, ale pro studijní účely si vytvoříme modul. Tento modul bude obsahovat funkci pro stažení kurzů z webu, funkce pro přepočítání z EUR a USD, potřebné aliasy. Modul nazveme **Kurzy**.

Nejprve si zobrazíme aktuální moduly začínající na 'k'.

```
PS C:\> Get-Module k* -list
```

Žádný takový modul neexistuje. V minulém díle jsme uváděli, že moduly jsou uloženy v adresářích uložených v proměnné prostředí **PSModulePath**.

```
PS C:\Scripts\kurzy> $env:PSModulePath -split ';'
C:\Documents and Settings\moravec\My
Documents\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
Dropbox:\PowerShell\Profile
```

První dvě cesty jsou standardní – pro konkrétního uživatele a pro všechny uživatele na počítači. Vidíte, že na svém počítači jsem přidal ještě třetí cestu a využívám Dropbox – tím mám zajištěno, že všechny moduly, které potřebuji na svých počítačích, mám stále dostupné. Ukažme si, co obsahuje první adresář:

```
PS C:\> ls ($env:PSModulePath -split ';')[0] |? { $_.PsIsContainer } |
Format-Wide Name -Column 3
Add-on.AuthoringToolkit Add-on.BlueConsole Add-on.ExpandAlias
Add-on.HelpBrowser Add-on.ModuleManagement Add-on.NumberedBookmarks
Add-on.PowerGUIOnline Add-on.ScriptEditorEssentials Add-
on.ScriptSigning
Add-on.TestAddOn adoLib Agent
CP2010 DemoPowerShell DotNet
FileSystem ISECreamBasic IsePack
MetaProgramming OracleClient OracleIse
```

```
PBM PoShTD PowerBoots
PowerShellPack PowerTab PSCodeGen
Pscx Pscx_old PSImageTools
PSRemoteRegistry PSRSS PSSystemTools
PSUserTools Repl Setup
ShareUtils ShowMbrs SMS2003
SQLIse SQLMaint SQLParser
SQLPSX SQLServer SSIS
TaskScheduler WPK
```

Poznámka: V tuto chvíli byste již měli rozumět celé konstrukci. Jediný možný zádrhel se vám může stát při zkoumání konstrukce `[? {$_.PsisContainer}]` – jedná se o filtrování pouze adresářů.

Každý modul je uložen v samostatném adresáři a je tvořen (minimálně) souborem stejného jména. Vytvořme si tedy základ pro náš modul.

```
PS C:\> mkdir (Join-Path ($env:PSModulePath -split ';')[0] Kurzy)
PS C:\> cd (Join-Path ($env:PSModulePath -split ';')[0] Kurzy)
PS C:\Documents and Settings\moravec\My
Documents\WindowsPowerShell\Modules\Kurzy> "" > Kurzy.psm1
PS C:\Documents and Settings\moravec\My
Documents\WindowsPowerShell\Modules\Kurzy> Get-Module k* -List
ModuleType Name ExportedCommands
-----
Script Kurzy {}
```

Nejprve jsme vytvořili adresář, poté soubor s příponou **PSM1** (PowerShell Module) a pak jsme si ověřili, že náš modul existuje a je „viditelný“. Samozřejmě zatím nic neumí. Přepíšeme tedy náš původní kód do trochu čitelnější podoby a vložíme jej do souboru **kurzy.psm1**.

```
function Get-CNBExchangeRate
{
$wc = New-Object Net.WebClient
$link =
'http://www.cnb.cz/cs/financni_trhy/devizovy_trh/kurzy_devizoveho_trhu/
denni_kurz.txt'
$file = 'C:\Scripts\kurzy\kurzy.txt'
$wc.DownloadFile($link, $file)
}
function Get-Kurz {
param([string]$kod)
Get-CNBExchangeRate
$file = 'C:\Scripts\kurzy\kurzy.txt'
$pattern = '\|(<kurz>\d{2}.\d{1,3})$'
gc $file |?
{ $_ -match $kod } |%
{ $_ -match $pattern | Out-Null; [double]$matches.kurz.replace(',',
'.') }
}

function eur {Get-Kurz EUR}
function usd {Get-Kurz USD}
```

Vše jsme rozdělili do dvou hlavních částí (funkcí). První funkce stáhne soubor s kurzy do daného adresáře a druhá provede vlastní konverzi na základě dané měny (určené parametrem **\$kod**). Funkce **eur** a **usd** nám slouží pouze pro rychlejší zobrazení požadované hodnoty.

```
PS C:\Documents and Settings\moravec\My
Documents\WindowsPowerShell\Modules\Kurzy> cd c:\
```

```
PS C:\> Import-Module Kurzy
PS C:\> usd
```

```
19,009
```

```
PS C:\> Get-Module k*
ModuleType Name ExportedCommands
```

```
-----
```

```
Script Kurzy {usd, eur, Get-Kurz, Get-CNBExchangeRate}
```

Po importu modulu můžeme jednoduchým příkazem (funkcí) zobrazit aktuální kurz. Všimněte si, že po dalším volání cmdletu **Get-Module** vidíme všechny funkce, které jsme vytvořili. Všechny jsou „exportované“ do aktuálního prostředí PowerShellu a můžeme tedy všechny použít. Pro naše potřeby, ale není nutné používat přímo funkci **Get-CNBExchangeRate**, protože tato funkce je volána uvnitř funkce **Get-Kurz** vždy, když je aktuálně zapotřebí. Proto využijeme možnost exportovat pouze funkce, které určíme pomocí cmdletu **Export-ModuleMember**. Jako poslední řádku našeho modulu přidáme

```
Export-ModuleMember Get-Kurz, eur, usd
modul odebereme, opětovně importujeme a vyzkoušíme, zda se změna
projevila.
```

```
PS C:\> rmo kurzy; ipmo kurzy; gmo k*
ModuleType Name ExportedCommands
```

```
-----
```

```
Script kurzy {usd, eur, Get-Kurz}
```

(je to vidět – opravdu mám rád aliasy Ve sloupci **ExportedCommands** nyní chybí „nechtěná“ funkce a pokud byste jí chtěli nyní použít, nebude pro vás dostupná.

Pro rychlou konverzi můžeme použít funkce **eur** a **usd** a v případě potřeby konverze jiné měny, můžeme použít funkci **Get-Kurz**. Možná by se nám ale hodilo, vytvořit si pro tuto funkci alias. To můžeme provést přímo v rámci modulu a tím zajistit, že alias se nám vytvoří při importu modulu. Trochu upravíme poslední přidanou řádku a místo ní vložíme následující tři:

```
Set-Alias kurz Get-Kurz
Export-ModuleMember -Function Get-Kurz, eur, usd
Export-ModuleMember -Alias kurz
Přidali jsme alias a navíc jej i určili pro export. Vše opět ověříme:
```

```
PS C:\> rmo kurzy; ipmo kurzy; gmo k* | fl
Name : kurzy
Path : C:\Documents and Settings\moravec\My
Documents\WindowsPowerShell\Modules\kurzy\kurzy.psm1
Description :
ModuleType : Script
Version : 0.0
NestedModules : {}
ExportedFunctions : {eur, Get-Kurz, usd}
ExportedCmdlets : {}
ExportedVariables : {}
ExportedAliases : kurz
```

Je vidět, že alias se nám opravdu objevil na seznamu. Nyní můžeme využít tento alias pro zjištění kurzu jakékoli jiné měny ze seznamu.

```
PS C:\> kurz jpy
22,534
```

Samozřejmě bychom mohli náš modul dále vylepšovat. Nyní nám například funguje pouze pokud nepoužíváme proxy server – v případě, že jej používáme (nebo nejsme připojeni na internet) modul nefunguje, mohli bychom chtít zobrazit seznam možných měn, atd. Pěkné domácí úkoly, že ano?

Závěrem bych chtěl upozornit na jeden užitečný Add-in do PowerGUI Script Editoru. Jmenuje se [Module Management](#) a slouží (mimo jiné) ke konverzi vašich současných skriptů do modulu. Návod na instalaci a použití je dostupný na stránce tohoto Add-inu. Pokud jej budete zkoušet, určitě si všimnete, že kromě souboru s příponou PSM1 (vlastní modul), vytvoří i soubor PSD1. Tento soubor se nazývá manifest a k čemu všemu se dá využít si ukážeme v příští části, kde naše putování po modulech ukončíme. Zároveň si také ukážeme binární moduly a podíváme se na jednu velice užitečnou vlastnost modulů – těšte se :).

PSAKE

Dnes se podíváme na modul, který se nechal inspirovat z jiných jazyků a používá se nejčastěji na automatizaci buildu (nebudu zde kostrbat překládat anglické dobře známé pojmy), ale stejně tak dobře najde uplatnění v situacích, kdy potřebujeme provést sekvenci na sobě závislých kroků. Modul si můžete stáhnout z <https://github.com/JamesKovacs/psake> a již je určité jasné, že se jmenuje *psake*.

Proč psake a ne msbuild?

Odpověď je jednoduchá. Pokud rádi editujete xml, pak vám nevádí práce s msbuildem. Pro ostatní psake (respektive PowerShell) nabízí komfort skriptovacího jazyka, kterého jste se v xml museli vzdát. Msbuild má samozřejmě také své přednosti, takže nejlepší volbou je kombinovat psake a msbuild dohromady – jak už to obvykle na světě bývá.

Krom toho je psake možné použít i pro administrátorské účely. Vše, co nabízí PowerShell, je přístupné v psake. Psake je totiž napsané v *PowerShellu*. Příklady snad řeknou více.

Jak psake rozchodit

Na domovské stránce <https://github.com/JamesKovacs/psake> stačí po kliknutí na tlačítko *Download* zvolit poslední release číslo 4.00. Tento zip soubor pak rozbalte na disk (dále předpokládám adresář `c:\psake`). Spusťte PowerShell konzoli a pokračujte příkazy:

```
PS> sl c:\psake
import-module .\psake.psm1
# psake modul je naimportovaný

Get-Help Invoke-psake -Full
```

Psake je dobře zdokumentovaný modul, takže poslední příkaz vypíše přehled parametrů použitelných při volání `Invoke-Psake`. Navíc ale obsahuje několik příkladů, na kterých je použití dobře patrné.

Jednoduchý příklad

Pro rychlé uvedení do problematiky si tu ukážeme jednoduchý příklad, jehož cílem je:

1. Adresář `d:\temp\code` přepokopíruje do `d:\temp\codebak` (pro jednoduchost ne rekurzivně).
2. Vylistuje seznam souborů z `d:\temp\codebak` uloží jej do `d:\temp\codebak\files.txt`.
3. Spustí příkaz na commit do VCS.

Z popisu jednotlivých kroků je jasné, že po sobě následují. Za jistých okolností se ale může hodit spustit kterýkoliv z příkazů samostatně. V psake skriptu by pak kroky byly popsány pomocí *tasks*:

```

task default -depends Full
task Full -depends Backup, ListFiles, Commit
task Backup {
    Write-Host Backup
    gci d:\temp\code | ? { !$_.PsIscontainer } | copy-Item -destination d:\temp\code
    bak
}
task ListFiles {
    Write-Host Files list
    gci d:\temp\codebak | Select -exp FullName | sc d:\temp\codebak\files.txt
}
task Commit {
    Write-Host Commit
    start-Process GitExtensions.exe -ArgumentList commit, d:\temp\codebak
}

```

Tento skript pak uložíme jako *psake-build.ps1* a spustíme. Pokud neuvedeme jméno tasku, psake použije task se jménem *default*:

```
PS> Invoke-psake -buildFile d:\temp\psake\psake-build.ps1
```

```
Executing task: Backup
```

```
Backup
```

```
Executing task: ListFiles
```

```
Files list
```

```
Executing task: Commit
```

```
Commit
```

```
Build Succeeded!
```

```
-----
Build Time Report
-----
```

Name	Duration
-----	-----
Backup	00:00:00.1396781
ListFiles	00:00:00.0548712
Commit	00:00:00.3255901
Full	00:00:00.5288634

```
Total: 00:00:00.6099274
```

A po skončení "buildu" se otevře okno GitExtensions se změnami do VCS. Slovo *build* se prolíná celým psake, ale znovu podotýkám, že nejde pouze o build. V příkladu jsme nic nepřekládali a nevyvíjeli.

V případě, že bychom potřebovali zavolat pouze některé tasky, specifikujeme je pod parametrem `-taskList`:

```
PS> Invoke-psake -buildFile d:\temp\psake\psake-build.ps1 -task Backup, Commit
```

Pokud dojde v některém kroku k chybě, následující kroky již nejsou vykonány. Jednoduše toho docílíme například kopírováním z neexistujícího adresáře: `$codeDir = 'd:\temp\doesntexist'`:

```
PS> Invoke-psake d:\temp\psake\psake-build.ps1
Executing task: Backup
Backup
psake-build.ps1:Cannot find path 'D:\temp\doesntexist' because it does not exist.
```

V případě, že bychom chtěli pokračovat navzdory chybám zaznamenaným při běhu tasku, můžeme toto u tasku určit parametrem `-ContinueOnError`:

```
PS> Invoke-psake d:\temp\psake\psake-build.ps1
Executing task: Backup
Backup
-----
Error in Task [Backup] Cannot find path 'D:\temp\doesntexist' because it does not exist.
-----
Executing task: ListFiles
....
```

Parametrizace

Psake skript je samozřejmě pořád skript napsaný v PowerShellu. Proto můžeme jména adresářů uložit do proměnné, aby byl skript přehlednější. Když se ale podíváme do některých psake skriptů, uvidíme konstrukci `properties { $var1 = 'value'; ... }`, co tedy použít?

Pokud jde o konstanty a nebudeme je chtít měnit, můžeme použít kteroukoliv z možností. Pokud bychom ale chtěli ve skriptu definovat default hodnotu nějaké proměnné, použijeme konstrukci `properties {...}`. Tak později můžeme default hodnotu přetížít pomocí parametru `-properties`. Soubor psake-build.ps1 by pak vypadal takto:

```
#fixní proměnná, nedá se měnit jinak než zápisem zde ve skriptu
$codeDir = 'd:\temp\code'
properties {
    #měnitelná proměnná
    $backupDir = 'd:\temp\codebak'
```

```

}

task default -depends Full

task Full -depends Backup, ListFiles, Commit

task Backup {
    Write-Host Backup
    gci $codeDir | ? { !$_.PsIscontainer } | copy-Item -destination $backupDir
}

...

```

A při volání bychom použili parametr `-properties`:

```

PS> Invoke-psake -buildFile d:\temp\psake\psake-build.ps1 -properties @{ backupdir
= 'd:\temp\otherbackdir' }

```

Všimněte si, že jako hodnotu předáváme *hashtable*, ne *scripblock*. Každá položka v hashtable specifikuje proměnnou, která bude vyhodnocena ve stejném scope jako `properties {...}` v psake skriptu (ale později).

Poznámka: výše uvedené není tak úplně pravda. I v případě, že do build skriptu napíšete `$backupdir = 'nejaka default cesta'` mimo blok `properties { ... }`, i této proměnné lze nastavit jiná hodnota z command line `Invoke-Psake ... -properties @{backupdir= 'jina cesta'}`. Spíše bych ji ale nedoporučil; v pozdějších verzích se může jiným způsobem pracovat se scope proměnných a skript by pak mohl přestat správně fungovat.

K čemu jsou Parameters

Psake ještě umožňuje specifikovat parametr funkce `Invoke-Psake` jménem `-parameters`. Jde opět o *hashtable* se stejnou strukturou jako `-properties`, tj. z každé dvojice *key-value* bude vytvořena proměnná. Tyto proměnné pak mohou být používány ve funkci `properties { ... }` v build skriptu – znamená to tedy, že tímto parametrizujeme skript. Z příkladu bude jasný rozdíl.

Předpokládejme, že build skript vypadá takto:

```

properties { $s = get-service $services }

task default

task stop { $s | stop-service -whatif }

task start { $s | start-service -whatif }

```

A jako vstupní parametr skriptu bychom poslali jméno/jména servis, které bychom chtěli nastartovat/zastavit:

```

PS> Invoke-psake -buildFile d:\temp\psake\psake-services.ps1 -task start -parameters @{ $services = 'W3SVC' }

```

V bloku `properties` jsme do proměnné uložili seznam servis odpovídajících vstupnímu parametru `$services`. Zde jsme mohli dodat jakoukoliv (složitější) inicializační logiku.

Někoho jistě napadne, jestli bychom mohli stejného efektu dosáhnout tím, že si nadefinujeme inicializační task a ten budeme volat před ostatními tasky, které na inicializaci závisí. Kód upraveného skriptu:

```
task default
properties { $services = "noservice" }
task init { $s = get-service $services }
task stop -depends init { Write-Host stop service $s; $s | stop-service -whatif }
task start -depends init { Write-Host start service $s; $s | start-service -whatif }
}
```

A skript bychom volali bez použití `-parameters`:

```
PS> Invoke-psake -buildFile d:\temp\psake\psake-services2.ps1 -task start -properties @{ $services = 'W3SVC' }
Executing task: init
Executing task: start
start service
psake-services2.ps1:Cannot bind argument to parameter 'Name' because it is null.
```

Z uvedeného je patrné, že myšlenka byla dobrá, ale psake na toto nebylo uzpůsobeno. Každý task (respektive jeho scriptblock) běží ve svém vlastním scope a proto proměnné přežívají pouze v rámci daného tasku. Mohli bychom sice takové chování obejít pomocí modifikátoru `script:`, ale taková úprava mění vnitřní stav modulu a proto zcela jistě není vhodná.

Psake pro vývojáře

Doposud jsme se bavili o psake pouze obecně a řekli jsme si většinu věcí, které může člověk potřebovat v případě, že si chce práci zautomatizovat a svázat úlohy pravidly. Programátor ocení funkci `exec`, která ukončí psake skript v případě, že program uvnitř skončí s chybou. Chyba je indikována návratovým kódem. Tělo je velmi jednoduché, můžeme se na něj podívat pomocí příkazu `Get-Content function:\exec`.

Task pro kompilaci solution vypadá s použitím `exec` velmi jednoduše:

```
$framework = '4.0'
...
task Build {
    exec { msbuild $slnPath '/t:Build' }
}
```

Msbuildu můžeme samozřejmě podstrčit další parametry, ale je vidět, že psake nám velmi usnadňuje práci s jeho zavoláním. Na začátku skriptu se říká, že se má použít msbuild pro verzi .NET 4.0. Psake si samo najde příslušný adresář a zajistí, že se spustí ten *naš* *správný* msbuild.

Jednoduchá programátorská automatizace buildu by pak mohla zahrnovat clean, build, spuštění testů, vytvoření databáze a nakopírování do release adresáře:

```
$framework = '4.0'
$sln = 'c:\dev\.....sln'
$outDir = 'c:\dev\...'
```

```

task default -depends Rebuild,Test,Out
task Rebuild -depends Clean,Build
task Clean {
    #exec { msbuild $slnPath '/t:Clean' }
    Write-Host Clean....
}
task Build {
    #exec { msbuild $slnPath '/t:Build' }
    Write-Host Build....
}
task Test {
    # run nunit console or whatever tool you want
    Write-Host Test....
}
task out {
    #gci somedir -include *.dll,*.config | copy-item -destination $outDir
    Write-Host Out....
}

```

Dá se toto ještě nějak zkrášlit? Ano – pro ty, kteří si rádi klikají (a mnohdy je to rychlejší, než vypisovat příkaz na příkazovou řádku) si můžeme vytvořit jednoduché GUI.

Psake do GUI

Pro vytvoření GUI použijeme WinForms. Nepůjde o krásu, ale o funkčnost, přizpůsobím tedy tomuto kód a maximálně jej zestručním.

PowerShell musí běžet v režimu **-STA**.

```

Add-type -assembly System.Windows.Forms
Add-type -assembly System.Drawing
if (! (get-module psake)) {
    sl D:\temp\psake\JamesKovacs-psake-b0094de\
    ipmo .\psake.psm1
}

$form = New-Object System.Windows.Forms.Form
$form.Text = 'Build'
$form.ClientSize = New-Object System.Drawing.Size 70,100

('build',10), ('test',30), ('out', 50) | % {

```

```

$cb = new-object Windows.Forms.CheckBox
$cb.Text = $_[0]
$cb.Size = New-Object System.Drawing.Size 60,20
$cb.Location = New-Object System.Drawing.Point 10,$_[1]
$form.Controls.Add($cb)
Set-Item variable:\cb$($_[0]) -value $cb
}

$go = New-Object System.Windows.Forms.Button
$go.Text = "Run!"
$go.Size = New-Object System.Drawing.Size 60,20
$go.Location = New-Object System.Drawing.Point 10,70
$go.add_Click({
    $form.Close()
    if ($cbbuild.Checked) { $script:tasks += 'Rebuild' }
    if ($cbtest.Checked) { $script:tasks += 'Test' }
    if ($cbout.Checked) { $script:tasks += 'Out' }
})
$form.Controls.Add($go)

$script:tasks = @()
$form.ShowDialog() | Out-Null
if ($script:tasks) {
    Invoke-psake -buildFile d:\temp\psake\psake-devbuild.ps1 -task $tasks
}

```

Na několika řádcích kódu jsme schopni si naklikat konfiguraci buildu. Nenechme se ale unést – build by měl být především automatický, pokud možno na jeden klik. Komplexní GUI s mnoha nastaveními nemusí být žádoucí!

Kód s importem modulu kontroluje, zda je psake již naimportované. Pokud bychom importovali modul podruhé, následující spuštění **Invoke-Psake** by skončilo chybou. Jde o problém samotného psake. Obecně by mělo jít moduly importovat vícekrát bez problémů.

Poznámka: práce se **\$script:tasks** mimo handler události vypadá těžkopádně. Proč nevolám **Invoke-Psake** přímo v handleru a předávám si seznam tasků ve zvláštní proměnné? Psake výsledek svého běhu (tabulku, časy, výpisy o běžících tasku) posílá do pipeline, aby bylo možné výstup přesměrovat do souboru. V handleru je tento výstup zpracováván jinak, neposílá se do hlavní pipeline a proto o výstup přijdeme. Jediné viditelné jsou výstupy z **Write-Host**, které jsou samozřejmě vypsány do konzole.

Změňte psake

Krátce si tu ukážeme, jak bez změny souboru s modulem můžeme změnit chování modulu. Jde o techniku používanou spíše zřídka. Mění totiž prostředí modulu. Bez dobrých znalostí vnitřní struktury modulu, může samozřejmě přestat pracovat korektně. Přesto – proč si nerozšířit znalosti?

Řekněme, že chceme v závěrečném souhrnu také zobrazovat aktuálního uživatele a jméno stroje. Postup je jednoduchý – je potřeba změnit funkci `Write-TaskTimeSummary`. Ve skutečnosti ji ale nezměníme:

```
PS> $module = Get-Module psake

PS> & $module { ${function:script:Write-TaskTimeSummaryBak} = (gi function:\Write-TaskTimeSummary).ScriptBlock }

PS> & $module { ${function:script:Write-TaskTimeSummary} = {
    . Write-TaskTimeSummaryBak
    "$env:USERNAME @$env:COMPUTERNAME"
}}
```

Čeho jsme tím dosáhli? Ve skriptu jsme vytvořili novou funkci `Write-TaskTimeSummaryBak` a do ní jsme zazálohovali aktuální obsah funkce `Write-TaskTimeSummary`. Poté jsme změnili definici funkce `Write-TaskTimeSummary` tak, aby volala zálohovanou funkci a na konec připojila řetězec se jménem uživatele a počítače.

Tento trik zřejmě často používat nebudete. Hodí se ale určitě znát obrat použitelný s jakýmkoliv modulem:

```
& (Get-Module mujmodul) { prikaz, který chci provést ve scope modulu }
```

Popsali jsme si základ práce s psake. Více informací lze najít především na <https://github.com/JamesKovacs/psake/wiki>. Namátkou se můžete těšit na tipy, jak nastavit akce před každým taskem a po každém tasku, podmínky nutné ke spuštění tasku, jak spustit psake z psake a další.

Ať se vám s psake dobře pracuje!

MODULY – DOKONČENÍ

V [minulé části](#) jsme si vytvořili vlastní modul sloužící ke zjišťování aktuálního kurzu. Dnes si ukážeme, jak tento modul ještě trochu vylepšit.

Manifest

Pokud se podíváme na informace o našem modulu, zjistíme, že toho vlastně až tak moc nevíme:

```
PS C:\> Get-Module Kurzy | fl *
```

```
ExportedCommands : {usd, eur, Get-Kurz}
Name              : kurzy
Path              : C:\Moduly\Kurzy\kurzy.psm1
Description       :
Guid              : 00000000-0000-0000-0000-000000000000
ModuleBase        : C:\Moduly\Kurzy
PrivateData       :
```

```

Version           : 0.0
ModuleType        : Script
AccessMode        : ReadWrite
ExportedFunctions  : {[eur, eur], [Get-Kurz, Get-Kurz], [usd, usd]}
ExportedCmdlets    : {}
NestedModules     : {}
RequiredModules   : {}
ExportedVariables  : {}
ExportedAliases    : {[kurz, kurz]}
SessionState      : System.Management.Automation.SessionState
OnRemove          :
ExportedFormatFiles : {}
ExportedTypeFiles  : {}

```

Pokud bychom takovýto modul získali na internetu, bylo by dobré o něm vědět trochu víc. K tomu účelu slouží právě manifest. Jedná se vlastně o hash tabulku (někdy se do češtiny překládá i jako asociativní pole), která obsahuje dvojice klíč-hodnota (key-value) určující vlastnosti manifestu.

Krátce k hash tabulce: Typickým případem je například dvojice hodnot jméno-věk, kdy k určitému člověku přiřadíme jeho věk, např. David – 33. V PowerShellu bychom tuto dvojici zapsali jako:

```

$h = @{
David = 33;
Martin = 34;
}

```

Zde jsme si uložili věk dvou lidí do hash tabulky a pro snazší práci tuto tabulku uložili do proměnné *h*. Nyní s ní můžeme dále pracovat.

```
PS C:\> $h
```

```

Name      Value
----      -
Martin    34
David     33

```

```
PS C:\> $h.GetType().FullName
System.Collections.Hashtable

```

```
PS C:\> $h.David
33

```

Jedna položka manifestu tedy vypadá například takto:

```
PS C:\Scripts > (Get-Content 'C:\Moduly\Pscx\Pscx.psd1')[2]
Author = 'PowerShell Community Developers'

```

Zde jsme z manifestu k modulu [PSCX](#) přečetli třetí řádek obsahující údaje o autorovi.

Další informace se můžete dozvědět například [v osmé části tohoto seriálu](#) nebo v nápovědě (**Get-Help about_Hash_Tables**). Pojdme ale zpátky k modulům.

Manifest můžete vytvořit několika způsoby (od manuálního zapsání v notepadu až po použití speciálního add-inu). My si vyzkoušíme použití cmdletu **New-ModuleManifest**.

```
PS C:\> New-ModuleManifest
```

```
cmdlet New-ModuleManifest at command pipeline position 1
Supply values for the following parameters:
Path: c:\Moduly\Kurzy\Kurzy.psd1
NestedModules[0]:
Author: David Moravec
CompanyName: PowerShell.cz
Copyright:
ModuleToProcess:
Description: Zjisteni aktualnich kurzu pro jednotlivy meny
TypesToProcess[0]:
FormatsToProcess[0]:
RequiredAssemblies[0]:
FileList[0]:
```

Vidíte, že některé z parametrů jsem přeskočil a vyplnil jsem pouze cestu k manifestu, jméno autora a popis modulu. PowerShell automaticky vytvořil soubor a uložil jej do námi definované cesty.

Poznámka: Nebudu zde celý soubor vypisovat. Jedná se zhruba o 100 řádek hodnot a komentářů. Pokud se chcete podívat, jak manifest vypadá, spusťte si předchozí cmdlet na vašem počítači. Zároveň si všimněte, že některé hodnoty, které jsem nezařadil, si PowerShell doplnil automaticky (např. GUID).

Nyní si můžeme opětovně pustit první příkaz dnešního článku.

```
PS C:\> Get-Module Kurzy | fl *
```

```
ExportedCommands      : {}
Name                  : Kurzy
Path                  : C:\Moduly\Kurzy\Kurzy.psd1
Description           : Zjisteni aktualnich kurzu pro jednotlivy meny
Guid                  : b0ec4496-18da-4e61-a37a-f98fc2b6e74a
ModuleBase            : C:\Moduly\Kurzy
PrivateData           :
Version               : 1.0
ModuleType            : Manifest
AccessMode            : ReadWrite
ExportedFunctions     : {}
ExportedCmdlets       : {}
NestedModules         : {}
RequiredModules       : {}
ExportedVariables     : {}
```

```
ExportedAliases      : {}
SessionState         :
OnRemove              :
ExportedFormatFiles  : {}
ExportedTypeFiles    : {}
```

A ihned vidíme změny. **ModuleType** se změnil na *Manifest* (z původního *Script*) a byly doplněny některé parametry. V manifestu můžeme určovat, jaké funkce či aliasy budeme exportovat, jestli potřebujeme pro běh našeho modulu nějaký jiný modul, jakou verzi PowerShellu potřebujeme, atd. Pro podrobnější zkoumání doporučuji podívat se do vytvořeného manifest souboru a sledovat komentáře.

Všimněte si jedné velké záludnosti! V seznamu exportovaných aliasů/funkcí nevidíme vůbec nic. Je to z toho důvodu, že jsme neuvedli žádnou hodnotu pro parametr **ModuleToProcess**. Můžeme to napravit ruční editací manifestu a opětovným importem modulu. Pak již bude vše v pořádku.

```
PS C:\> (Get-Module Kurzy).ExportedCommands
```

Name	Value
----	-----
usd	usd
eur	eur
Get-Kurz	Get-Kurz

Pokud editujete manifest ručně, může se vám hodit cmdlet **Test-ModuleManifest**. Ten provede kontrolu správnosti manifestu a pokud neohlásí chybu je vše v pořádku (alespoň z pohledu manifestu samotného).

```
PS C:\Scripts > Test-ModuleManifest -Path
'C:\Moduly\Kurzy\Kurzy.psd1'
```

```
Test-ModuleManifest : The 'C:\Moduly\Kurzy\Kurzy.psd1' module cannot
be imported because its manifest contains one or more members that
are not valid. The valid manifest members are ('ModuleToProcess',
'NestedModules', 'GUID', 'Author', 'CompanyName', 'Copyright',
'ModuleVersion', 'Description', 'PowerShellVersion',
'PowerShellHostName', 'PowerShellHostVersion', 'CLRVersion',
'DotNetFrameworkVersion', 'ProcessorArchitecture',
'RequiredModules', 'TypesToProcess', 'FormatsToProcess',
'ScriptsToProcess', 'PrivateData', 'RequiredAssemblies',
'ModuleList', 'FileList', 'FunctionsToExport', 'VariablesToExport',
'AliasesToExport', 'CmdletsToExport'). Remove the members that are
not valid ('CompanyNme'), then try to import the module again.
At line:1 char:20
+ Test-ModuleManifest <<<< -Path 'C:\Moduly\Kurzy\Kurzy.psd1'
    + CategoryInfo          : InvalidData: ( 'C:\Moduly\Kurzy.psd1:String)
[Test-ModuleManifest], InvalidOperationException
    + FullyQualifiedErrorId :
```

```
Modules_InvalidManifestMember,Microsoft.PowerShell.Commands.TestModuleManifestCommand
```

ModuleType	Name	ExportedCommands
Script	kurzy	{usd, eur, Get-Kurz}

Pokud se pozorně podíváte do chybového výpisu, všimnete si, že zhruba v polovině se objevilo

```
Remove the members that are not valid ('CompanyNme'), then try to import the module again.
```

Opravdu – ručně jsem změnil správnou hodnotu **CompanyName** na špatnou **CompanyNme**. Po opravě už je manifest otestován správně.

Závěrem

Nakonec bych vám ještě rád ukázal několik zajímavých modulů.

PSCX – PowerShell Community Extensions dostupné na [Codeplexu](http://codeplex.com/powershellcommunityextensions). Jedná se o velice pěknou sadu 87 cmdletů:

```
PS C:\> gcm -Module pscx -CommandType cmdlet | Format-Wide
```

```
Add-PathVariable
Clear-MSMQueue
ConvertFrom-Base64
ConvertTo-Base64
ConvertTo-MacOs9LineEnding
ConvertTo-Metric
ConvertTo-UnixLineEnding
ConvertTo-WindowsLineEnding
Convert-Xml
Disconnect-TerminalSession
Expand-Archive
Export-Bitmap
Format-Byte
Format-Hex
Format-Xml
Get-ADObject
Get-AdoConnection
Get-AdoDataProvider
Get-AlternateDataStream
Get-Clipboard
Get-DhcpServer
Get-DomainController
Get-DriveInfo
Get-EnvironmentBlock
```

Get-FileTail
Get-FileVersionInfo
Get-ForegroundWindow
Get-Hash
Get-HttpResource
Get-LoremIpsum
Get-MountPoint
Get-MSMQueue
Get-OpticalDriveInfo
Get-PathVariable
Get-PEHeader
Get-Privilege
Get-PSSnapinHelp
Get-ReparsePoint
Get-ShortPath
Get-TabExpansion
Get-TerminalSession
Get-TypeName
Get-Uptime
Import-Bitmap
Invoke-AdoCommand
Invoke-Apartment
Join-String
New-Hardlink
New-Junction
New-MSMQueue
New-Shortcut
New-Symlink
Out-Clipboard
Ping-Host
Pop-EnvironmentBlock
Push-EnvironmentBlock
Read-Archive
Receive-MSMQueue
Remove-AlternateDataStream
Remove-MountPoint
Remove-ReparsePoint
Resolve-Host
Send-MSMQueue
Send-SmtpMail
Set-BitmapSize
Set-Clipboard
Set-FileTime
Set-ForegroundWindow
Set-PathVariable
Set-Privilege
Set-VolumeLabel
Skip-Object
Split-String

Start-TabExpansion
Stop-TerminalSession
Test-AlternateDataStream
Test-Assembly
Test-MSMQueue
Test-Script
Test-UserGroupMembership
Test-Xml
Unblock-File
Write-BZip2
Write-Clipboard
Write-GZip
Write-Tar
Write-Zip

Rozhodně je vyzkoušejte, některé cmdlety jsou hodně povedené.

SQLPSX – SQL Server PowerShell Extensions (opět na [Codeplexu](#)) slouží k práci s SQL serverem. Velké plus vidím v integraci s ISE, kde můžete použít například Object Browser.

WPK – jako součást [PowerShellPacku](#), o tomto module jsem již psal v jedné z předchozích částí.

Tímto bychom ukončili naše putování po modulech. Příště se podíváme na procházení logů a jejich filtrování pomocí regulárních výrazů.

PROHLEDÁVÁME TEXT

Než začneme s dnešním tématem, uděláme si malou exkurzi do novinek ze světa PowerShellu. V dubnu se v Las Vegas konal další ročník konference [TEC](#) (The Experts Conference), její součástí byla i první [PowerShell Deep Dive](#) konference. Na této konferenci byla [oznámena](#) pěkná novinka - specifikace PowerShellu byla [uvolněna](#) pod licencí [Microsoft Community Promise](#). Na první pohled nic zajímavého, ale pokud se do specifikace podíváte, najdete v ní mnoho zajímavých informací o fungování PowerShellu.

Základní porovnávání textu

V posledním dílu jsem slíbil, že se dnes podíváme na regulární výrazy. Musím se ale předem omluvit. Zamýšlený úvod se rozrostl více než jsem předpokládal a proto si regulární výrazy necháme až do dalšího pokračování.

Nejdřív se tedy podíváme na jednoduché a rychlé porovnávání textu. S tímto typem porovnávání si vystačíme ve většině případů.

Poznámka: V tomto článku budu mluvit o porovnání textů, ale vše samozřejmě platí i pro jiné datové typy.

PowerShell nám nabízí základní operátory porovnání, jsou to např. *-eq*, *-gt*, *-like*, *-contains*. Nebudu je vyjmenovávat všechny, jejich seznam najdete v tématické nápovědě

[about Comparison Operators](#). Než si ukážeme jednoduchý příklad, je potřeba zmínit ještě jednu věc. Pro většinu operátorů platí, že pokud je na levé straně porovnání (vstup) jediná hodnota, výstupem je true nebo false. Pokud je na vstupu více hodnot, výstupem jsou všechny vstupy splňující podmínku.

```
PS C:\> 'PowerShell' -eq 'powershell'
True
PS C:\> 'PowerShell','VBS','Perl','bash' -eq 'powershell'
PowerShell
PS C:\> 'PowerShell','VBS','Perl','bash' -ne 'powershell'
VBS
Perl
Bash

PS C:\> 'PowerShell' -ceq 'powershell'
False
PS C:\> 'PowerShell','VBS','Perl','bash' -cne 'powershell'
PowerShell
VBS
Perl
Bash
PS C:\> 'PowerShell','VBS','Perl','bash' -ceq 'powershell'

PS C:\> 1..10 -gt 5
6
7
8
9
10

PS C:\> 1..10 -le 5
1
2
3
4
5
```

První tři příkazy jsou jednoduché, zjišťujeme, jestli je v zadaném vstupu text „PowerShell“. Další tři příkazy již porovnávají text včetně kontroly velkých/malých písmen (c v názvu od case-sensitive) a proto nám první z nich vrátí hodnotu false. Poslední dva příkazy ukazují porovnání pomocí operátorů větší než/menší rovno.

Poznámka: Velká diskuse se strhla kvůli pojmenování operátorů písmeny a nepoužití „zažítých“ operátorů jako je například ‘>’. Níže uvádím citaci z knihy [PowerShell in Action](#) od Bruce Payetta, který je jedním z designérů jazyka PowerShellu:

Let's talk about the most contentious design decision in the PowerShell language. And the winner is: why the heck did we not use the conventional symbols for comparison like “>”, “>=”, “<”, “<=”, “==”, and “!=” ? My, this was a touchy issue. The answer is that the “>” and “<” characters are used for output redirection. Since PowerShell is a shell and all

shell languages in the last 30 years have used “>” and “<” for I/O redirection, people expected that PowerShell should do the same. During the first public beta of PowerShell, this topic generated discussions that went on for months. We looked at a variety of alternatives, such as modal parsing where sometimes “>” meant greater-than and sometimes it meant redirection. We looked at alternative character sequences for the operators like “:>” or “->”, either for redirection or comparison. We did usability tests and held focus groups, and in the end, settled on what we had started with. The redirection operators are “>” and “<”, and the comparison operators are taken from the UNIX test(1) command. We expect that, since these operators have a 30-year pedigree, they are adequate and appropriate to use in PowerShell. (We also expect that people will continue to complain about this decision, though hopefully not for 30 more years.)

Mě osobně použité řešení vyhovuje. Přemýšlení, jestli ‘=’ je porovnání nebo přiřazení a jestli ‘<’ je to same jako ‘!’ mě v jiných programovacích jazycích trochu vadí (i když chápu, že vývojář, který píše každý den v C# nemusí mou radost/starost sdílet).

Dalšími operátory, které se hodí pro porovnání textu, jsou *like* a *match*. *Like* funfuje tak, jak je většina z nás zvyklá z operačního systému. Využívá znaků * a ? – použití je stejné již od dob MS-DOSu. Všichni už určitě někdy psali *.txt. Méně známá je možnost použití hranatých závorek pro výčet, např. [a-d] znamená jakýkoli ze znaků a, b, c, d. Ukažme si vše opět na příkladu.

```
PS C:\> 'PowerShell' -like 'powershell'
True
PS C:\> 'PowerShell' -clike 'powershell'
False
PS C:\> 'PowerShell' -like 'shell'
False
PS C:\> 'PowerShell' -like '*shell'
True
PS C:\> 'PowerShell' -like 'P*shell'
True
PS C:\> 'PowerShell' -like 'P[o-q]*shell'
True
PS C:\> 'PowerShell' -like 'P[p-q]*shell'
False
PS C:\> 'PowerShell' -like 'P[awpo]*shell'
True
PS C:\> 'PowerShell' -like 'P[awp]*shell'
False
```

Nejprve porovnáваме *like* a case-sensitive *like*. Třetí ař pátý příklad ukazují (očekávané) použití hvězdičky. Další příklady už používají zmiňovaný operátor rozsahu. Pokud bychom trvali na velikosti písmen, mohli bychom slovo PowerShell porovnat například takto.

```
PS C:\> 'PowerShell' -clike '[A-Z]?[aeiouy][rs][RS]hell*'
True
```

Z předchozího jsou vidět dvě věci – otazník opravdu zastupuje jeden libovolný znak a hvězdička (na konci) znamená ve skutečnosti i nulový výskyt znaku. Vše si ještě potvrdíme závěrečnou ukázkou.

```
PS C:\> '' -like '*'
True
PS C:\> '' -like '?'
False
```

Operátor *match* porovnává vstupní texty pomocí regulárních výrazů. K nim se dostaneme příště, nyní si ještě ukážeme jeden užitečný cmdlet.

Select-String

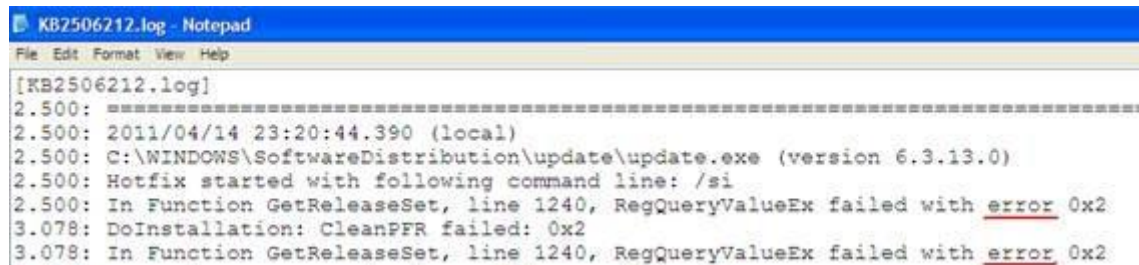
Pokud chcete ve vstupním textu (nebo souboru) vyhledávat vzory textu, je [Select-String](#) ideálním pomocníkem. Zkusme se podívat, jestli se v log souborech objevují hlášení o chybách.

```
PS C:\temp> ls kb25????2.log | Select-String 'error' -SimpleMatch

KB2506212.log:6:2.500: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:8:3.078: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:34:3.484: KB2506212 Setup encountered an error: The
update.ver file is not correct.
KB2506212.log:48:3.828: Update.exe extended error code = 0xf200
KB2506212.log:53:1.875: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:55:2.234: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:61:3.765: Update.exe extended error code = 0xf201
KB2506212.log:67:1.922: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:86:2.328: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2506212.log:112:2.687: KB2506212 Setup encountered an error: The
update.ver file is not correct.
KB2506212.log:113:2.687: KB2506212 Setup encountered an error: The
update.ver file is not correct.
KB2506212.log:114:2.687: KB2506212 Setup encountered an error: The
update.ver file is not correct.
KB2506212.log:115:2.687: KB2506212 Setup encountered an error: The
update.ver file is not correct.
KB2508272.log:7:1.906: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
KB2508272.log:38:2.313: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
```

KB2508272.log:53:2.406: KB2508272 Setup encountered an error: The update.ver file is not correct.

Pokud vstupní soubory obsahují hledaný text, je vypsána každá řádka, kde se vyskytuje, včetně čísla této řádky. Čistě pro kontrolu, vstupní soubor vypadá takto:



```
KB2506212.log - Notepad
File Edit Format View Help
[KB2506212.log]
2.500: =====
2.500: 2011/04/14 23:20:44.390 (local)
2.500: C:\WINDOWS\SoftwareDistribution\update\update.exe (version 6.3.13.0)
2.500: Hotfix started with following command line: /si
2.500: In Function GetReleaseSet, line 1240, RegQueryValueEx failed with error 0x2
3.078: DoInstallation: CleanPFR failed: 0x2
3.078: In Function GetReleaseSet, line 1240, RegQueryValueEx failed with error 0x2
```

Častokrát nám ovšem stačí pouze vědět, že v souboru je *nějaká* chyba – soubor můžeme pak zkopírovat či poslat mailem na další kontrolu. Zkusme hledat čas, kdy začal zápis do logu z předchozího obrázku (23:20:44)

```
PS C:\temp> ls kb*.log | Select-String '23:20:44' -SimpleMatch -
Quiet
```

```
False
False
False
False
True
False
False
False
False
False
False
False
```

J Parádní výsledek, kdy jsme se dozvěděli, že ve vstupních souborech je jeden, který vyhovuje našemu zadání. Abychom se dozvěděli i jméno tohoto souboru, musíme náš příkaz trochu upravit:

```
PS C:\temp> ls kb*.log | Where-Object { Select-String '23:20:44' -
Input $_ -Simple -Quiet } | Select Name
```

```
Name
----
```

```
KB2506212.log
```

Seznam vstupních souborů jsme filtrovali pomocí **Where-Object**, pokud soubor obsahoval hledaný text, tzn. cmdlet **Select-String** vrátil hodnotu true, vypsali jsme jméno souboru.

Pokud jsme našli potřebný soubor, můžeme se dále podívat, kde se hledaný text vyskytuje. Velice užitečným parametrem je *Context*. Udáváme jím, kolik řádek před/za hledaným, chceme zobrazit ve výpisu.

```
PS C:\temp> Select-String -Path kb2506212.log -Pattern '23:20:44' -SimpleMatch -Context 2
```

```
kb2506212.log:1:[KB2506212.log]
kb2506212.log:2:2.500:
=====
> kb2506212.log:3:2.500: 2011/04/14 23:20:44.390 (local)
kb2506212.log:4:2.500:
C:\WINDOWS\SoftwareDistribution\update\update.exe (version 6.3.13.0)
kb2506212.log:5:2.500: Hotfix started with following command line:
/si
```

```
PS C:\temp> Select-String -Path kb2506212.log -Pattern '23:20:44' -SimpleMatch -Context 1, 5
```

```
kb2506212.log:2:2.500:
=====
> kb2506212.log:3:2.500: 2011/04/14 23:20:44.390 (local)
kb2506212.log:4:2.500:
C:\WINDOWS\SoftwareDistribution\update\update.exe (version 6.3.13.0)
kb2506212.log:5:2.500: Hotfix started with following command line:
/si
kb2506212.log:6:2.500: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
kb2506212.log:7:3.078: DoInstallation: CleanPFR failed: 0x2
kb2506212.log:8:3.078: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
```

Jako hodnoty parametru můžeme uvést jedno nebo dvě čísla. Pokud uvedeme jedno, PowerShell zobrazí daný počet řádek před a za hledaným textem. V případě, že uvedeme čísla dvě, platí první pro počet řádek před hledaným textem a druhé pro počet řádek za ním.

Select-String může porovnávat vstupní text i pomocí regulárních výrazů. Jistě jste si všimli, že jsem u všech příkladů použil parametr *SimpleMatch* – ten říká, že **nechci** pro porovnání regulární výrazy použít. Na konci příštího dílu si vyzkoušíme použití **Select-String** bez parametru *SimpleMatch*.

Závěrem

Než úplně skončíme, vrátíme se ještě na chvíli k porovnávání pomocí operátorů. Zajímavý je operátor *is*. Tím testujeme jakého typu je vstupní parametr.

```
PS C:\temp> 'PowerShell' -is 'System.String'
True
```

Toho můžeme využít pro pěkný trik. Pokud žádáme po uživateli jméno heslo, můžeme v PowerShellu použít cmdlet Get-Credentials. Pokud máme funkci a nevíme, jestli je vstupem pouze uživatelské jméno nebo již credentials, lze použít následující trik (v PowerShellu v2 bychom mohli situaci řešit pomocí advanced funkcí elegantněji, teď mi jde čistě o ukázkou operátoru is):

```
function Get-Cred
{
    param ($cred)
    Write-Host "Before: $($cred.GetType().FullName)"
    if ($cred -is 'System.String')
    {
        $cred = Get-Credential -Credential $cred
    }
    Write-Host "After: $($cred.GetType().FullName)"
}
```

V prvním případě zadáváme jako vstup text:

```
PS C:\> Get-Cred -cred aaa
Before: System.String
After: System.Management.Automation.PSCredential
```

Ve druhém už předáváme credentials

```
PS C:\> $c = Get-credential -credential bbb
PS C:\> Get-Cred -cred $c
Before: System.Management.Automation.PSCredential
After: System.Management.Automation.PSCredential
```

POWERSHELL Z POHLEDU PROGRAMÁTORA

Na úvod se musím zmínit o největší události za posledních pár měsíců, alespoň z mého pohledu: kniha [PowerShell In Action](#) (autorem je Bruce Payette) je konečně k dostání i v papírové podobě. Pokud to s PowerShellem myslíte opravdu vážně, tato kniha by už měla ležet ve vašem nákupním košíku.

Při jedné kratší diskuzi na Twitteru jsem se dozvěděl, že někteří PowerShell nepotřebují (nebo si to alespoň myslí :) a případně, že PowerShell má obskurní syntaxi a napsaný [Y combinator](#) je snad nečitelnější, než kdyby to autor napsal v peru. A další otázka se týkala tutoriálu k PowerShellu.

Proto jsem se rozhodl, že PowerShell zkusím představit očima programátora. Jako inspirace mi posloužilo Augiho povídání o [javascriptu](#). Dnešní článek tedy bude spíše teoretický, ale budu se snažit jej co nejvíce odlehčit příklady. Cílem je ukázat základní podobnosti a rozdíly mezi PowerShellem a běžným programovacím jazykem. Více informací se dá najít buď v helpu PowerShellu, nebo například ve výše uvedené knize.

A zde je obsah dnešního článku:

- [Framework](#)
- [Typy](#)
- [Operátory](#)
- [Funkce](#)
- [Pipeline](#)
- [OOP](#)
- [Generiky](#)
- [Scope, platnost proměnných](#)
- [Vyhodnocování výrazů](#)

Framework

PowerShell staví na .NET frameworku. Proto ti, kteří s .NET frameworkem pracují, získávají jednoznačnou výhodu – mohou samozřejmě využít při skriptování svých znalostí. PowerShell je zkompileovaný pro .NET 2.0/3.5. Dá se ale dosáhnout toho, že poběží v .NET 4 runtime. Jak? [Otázka na SO](#) obsahuje mnoho tipů. Ve zkratce nejjednodušší způsob je tento:

1. Otevřete PowerShell config. Config pro ISE se jmenuje `powershell_ise.exe.config` a otevřete jej příkazem `notepad $pshome\powershell_ise.exe.config`. Je možné, že zatím neexistuje a bude jej potřebovat vytvořit.
2. Do configu přidejte tento záznam:
3. `<startup useLegacyV2RuntimeActivationPolicy="true">`
4. `<supportedRuntime version="v4.0"/>`
5. `<supportedRuntime version="v2.0.50727" />`
`</startup>`
6. A restartujte ISE.

Obdobně pro PowerShell konzoli můžete změnit/vytvořit soubor `$pshome\powershell.exe.config`. Musíme ale počítat s tím, že start PowerShellu je při runtime .NET 4 daleko pomalejší.

Typy

Vzhledem ke svým základům – .NET frameworku – se nedá vymezit pevná sada typů, se kterou PowerShell pracuje. Místo toho můžeme poukázat na nejběžněji používané typy, které se většinou vyskytují také v jiných programovacích jazycích.

- *číslo* – reprezentované typy `[int]`, `[int64]`, `[float]`, `[decimal]`, ...
- *bool* – tedy hodnota ano/ne. K těmto hodnotám přísluší konstanty `$true` a `$false`.
- *řetězec* – řetězce je možné zapisovat do apostrofů i do uvozovek a mohou se roztáhnout přes více řádek.
- *scriptblock* – je v podstatě anonymní funkce. Zapisuje se do složených závorek. Příklad: `{ Write-Host toto je test }` je anonymní funkce, která nic nevrací, pouze vypíše nějaký text na obrazovku.
- *hashtable* – je reprezentovaná starou známou `System.Collections.Hashtable`.
- *pole* – je pole obecných objektů, tj. `System.Object[]`.

Bool

Typ *bool* není potřeba popisovat. Proto zde zmíním pouze přetypování na *[bool]*:

```
[bool]1          # true
[bool]0          # false
[bool]''         # false
[bool]'a'        # true
[bool]1.1        # true
[bool]$null      # false
[bool][bool]     # true
[bool](1..10|Where-Object{$_ -gt 100 }) # false (1)
[bool](@())      # false (2)
[bool](@{})      # true
```

V praxi je důležité přetypování (1) a (2). Úzce souvisí s pipeline – prázdná sekvence se konvertuje na *\$false*, neprázdná pak na *\$true*:

```
if (Get-ChildItem d:\temp) {
    Write-Host Dir d:\temp is not empty
}
```

Řetězec

Jaký je rozdíl mezi apostrofy a uvozovkami? Ukážeme na příkladu:

```
$today = Get-Date
Write-Host "Dnes je $today" # vypíše Dnes je 05/27/2011 18:30:29
Write-Host 'Dnes je $today' # vypíše Dnes je $today
```

V případě, že použijeme uvozovky, proměné se vyhodnotí a poté se celý řetězec předá jako parametr cmdletu *Write-Host*. V případě apostrofů se ale řetězec vezme jako takový a vyhodnocení neprobíhá.

Zde bych rád upozornil na možnou chybu při formátování. Srovnajte:

```
$today          # 27. května 2011 18:30:29
Write-Host $today # 27.5.2011 18:30:29
"$today"        # 05/27/2011 18:30:29
```

Obdobná pravidla a nesrovnalosti platí například i pro floaty. Naštěstí se s tímto problémem dá dobře žít, pokud o něm víme.

Formátovat (datum) můžete standardními prostředky, které nabízí .NET, nebo samotný PowerShell. Konkrétně pro datum můžete přímo určit, v jakém formátu se má výstup vrátit takto:

```
Get-Date -Format yyyy-MM-dd          # 2011-05-27
(Get-Date).ToString('yyyy-MM-dd')    # 2011-05-27
```

Scriptblock

V některých jazycích jsou považovány funkce za jeden ze základních typů. Stejně tak je tomu i v PowerShellu, kde anonymní funkce je reprezentovaná typem *scriptblock*. Scriptblock může vyžadovat parametry, nebo být bezparametrický:

```
$listC = { Get-ChildItem c:\ }
# anonymní funkci zavoláme & operátorem
& $listC

$listCWithFilter = { param([string]$filter) Get-ChildItem c:\ -filter
$filter }
& $listCWithFilter *.txt
```

Hlavní využití scriptblocku je ovšem ve funkcích/cmdletech. Typickým příkladem jsou cmdlety `Where-Object` or `Foreach-Object`, kde scriptblock předáváme jako parametr.

```
Get-ChildItem c:\ | Where-Object {$_.PsIsContainer} | Foreach-Object
{$_.LastWriteTime}
```

Přetypováním na *[string]* získáme kód scriptblocku. Pokud bychom naopak měli kód a potřebovali vytvořit scriptblock, pomůže nám metoda `[scriptblock]::Create`.

```
& ([scriptblock]::Create('Write-Host (get-date)'))
```

Pokud se při vytváření scriptblocků neobejdete bez uzávěrů, použijte metodu

`GetNewClosure`:

```
$scriptblocks = Get-ChildItem |
    Foreach-Object {
        $item = $_
        { Write-Host Name is $item.Name }
    }
$scriptblocks | Foreach-Object { & $_ }

# versus
$scriptblocks = Get-ChildItem |
    Foreach-Object {
        $item = $_
        { Write-Host Name is $item.Name }.GetNewClosure()
    }
$scriptblocks | Foreach-Object { & $_ }
```

Hashtable

Hashtable pravděpodobně každý .NET vývojář zná hlavně z dob před generikami. Princip je jednoduchý: jde o kolekci, která udržuje dvojice key-value. Vytvoříme ji pomocí literálu `@{}`. Při přístupu použijeme hranaté závorky, jak jsme zvyklí, nebo tečkové notace (usnadnění PowerShellu):

```
$translation = @{monday='po'; tuesday='ut'}
$translation.wednesday = 'st'
$translation['thursday'] = 'ct'
$translation.ContainsKey('monday')      # true
$translation.ContainsKey('MONDAY')      # true
```

Všimněte si, zápisu `monday='po'`. Levá strana před rovnítkem je uvedena bez apostrofů/uzavírek. Pokud PowerShell může levou stranu zkonvertovat na nějaký základní typ, udělá to. Jinak považuje výraz za řetězec.

Poslední příklad také připomíná, že PowerShell je obecně *case-insensitive* a jak vidno, platí to i pro hashtable.

Jedna z ošklivých věcí v PowerShellu je procházení key-value párů. Je potřeba si nejdříve vrátit enumerátor a ten poslat do pipe. Důvodem tohoto neintuitivního chování je zřejmě skutečnost, že většina uživatelů by netušila, s jakým typem vlastně v pipeline budou pracovat (tj. neočekávali by property key a value).

```
$translation.GetEnumerator() | % { "{0} - {1}" -f $_.Key, $_.Value }
```

Čas od času se může také hodit sčítání objektů typu hashtable:

```
$prvni = @{1 = 'jedna'; 2='dva' }
$druha = @{3 = 'tri' }
$prvni + $druha
```

Hashtable se typicky v PowerShellu používá na dvě věci: *splatting* a vytváření custom objektů:

```
$parameters = @{Path='c:\'; Filter='*.txt' }
Get-ChildItem @parameters | # splatting
Foreach-Object {
    new-object PsObject -property @{
        Name=$_.FullName;
        Size=$_.Length } # property pro custom objekt
}
```

Velmi stručně řečeno se splatting hodí v situacích, kdy při psaní kódu ještě nemusíme vědět, s jakými parametry budeme chtít daný cmdlet zavolat. Upozorňuji, že nejde o hodnoty parametrů (argumenty), ale skutečně výčet použitých parametrů.

```
function GetItems { param($p) Get-ChildItem @p } $parameters =
@{Path='c:\'} if ($env:computername -eq 'qa-test') { $parameters['filter']
= '*.xls' } GetItems $parameters
```

Proměnná `$parameters` udržuje seznam parametrů a jejich hodnot pro volání `Get-ChildItem`. A pouze na testovacím stroji vylistuje jen xls soubory.

Array

Pole hraje jednu z klíčových rolí v PowerShellu. Může obsahovat jakýkoliv objekt, nekontroluje se typ. Pole vytvoříme pomocí literálu `@()`.

V praxi se ještě často používá operátor čárky. Ten nám také vrátí pole. Jen je potřeba myslet na prioritu, s jakou jsou operátory vyhodnocovány.

```
$arr1 = @('test') # jednoprvkové pole
$arr2 = , 'test' # jednoprvkové pole, stejně jako $arr1
$arr3 = 'test1', 'test2' # dvouprvkové pole
$fail = 1, 4-2 # chyba; musí být 1, (4-2)
```

Pro upřesnění: při použití `@(...)` PowerShell zkontroluje, jestli objekt uvnitř závorek je pole. Pokud ano, vrátí ono pole. Pokud není, zabalí objekt mezi závorkami do pole. Proto vícenásobná aplikace je zbytečná:

```
$arr1 = @('test')
$arr2 = @($arr1) # arr1 a arr2 jsou ekvivalentní
```

```
$arr3 = @(@(@('test')) # opět stejné jako arr1
```

Pole se dají počítat. Následující příklad ukazuje, jak při sčítání polí dostat očekávané výsledky:

```
$arr1 = @('test')
$arr2 = @('test2', 'test3')

$result1 = $arr1 + $arr2 # co je uvnitř? (1)
$result2 = , $arr1 + , $arr2 # a co tady? (2)

# otestujeme
$result1[1][1] # vrátí e
$result2[1][1] # vrátí test3
```

Je patrné, že při sčítání dvou polí dojde k připojení prvků z druhého pole (1). Pokud bychom chtěli pole nechat izolované, je potřeba je zabalit jako jednoprvkové pole (pomocí operátoru čárky) a tato zabalená pole sečíst (2).

Dynamičnost

PowerShell na první pohled působí jako dynamicky typovaný jazyk:

```
function Get-Length{ param($object) $object.Length }

Get-Length 'test'
Get-Length (Get-Item $profile)
```

V obou případech voláme property `Length`. při prvním volání předáváme řetězec, v druhém *FileInfo*. Ve skutečnosti je ale PowerShell staticky typovaný.

PowerShell na pozadí každý objekt zabalí do instance typu `PSObject`. Zde se udržuje seznam dostupných metod a properties objektu. Můžeme se na něj podívat takto:

```
$xml = [xml]'<root><test/></root>'
$xml.PsObject # vrátí obalující objekt
$xml.PsBase    # vrátí proxovaný objekt
```

Díky tomuto obalení PowerShell může přidat další property k danému objektu:

```
$xml.root
```

Property `root` ve třídě `XmlDocument` samozřejmě obsažená není. Kvůli zjednodušení práce s XML ji tam PowerShell dodal, aby se dalo do XML dotazovat pomocí tečkové notace. Záznam o této property pak získáme takto:

```
$xml.PsObject.Properties | Where-Object {$_.Name -eq 'root'}
```

Operátory

Aritmetické

Mezi základní aritmetické operátory patří `+` `-` `*` `/` `%`. Toto zřejmě nikoho nepřekvapí:

```
$a = 2
```

```

$b = 3
$a + $b
$a / $b      # pozn. (1)
$a / $a      # pozn. (1)
$a % $b
'test_' * $b # pozn. (2)
$a + "10"    # pozn. (3)
'test_' * "10" # pozn. (3)

```

Za povšimnutí stojí několik zjištění:

- (1) PowerShell inteligentně pozná, jestli již je potřeba pracovat s plovoucí čárkou, nebo ne. Proto vrátí pro `$a/$b` typ *[double]* a pro `$a/$a` typ *[int]*.
- (2) Pokud použijeme operátor násobení s řetězcem na levé straně a číslem na pravé, PowerShell provede zopakování řetězce.
- (3) PowerShell používá poměrně mocný systém konverzí. Proto zafunguje nejen výraz `$a + "10"`, ale i `'test_' * "10"`. PowerShell neumí násobit dva řetězce, proto automaticky zkonvertuje druhý řetězec na číslo a provede násobení řetězce číslem.

Logické

Syntaxe logických operátorů poněkud trpí nečitelností: `-and` `-or` `-not` `-xor`. Naštěstí alespoň operátor `-not` můžeme psát jako vykřičník:

```

-not $true # je false
! $true    # opět false

```

U logických operátorů se uplatní zkrácené vyhodnocování, tj. pokud nalevo od `-and` je `$false`, zbytek se nevyhodnocuje. Obdobně pro `-or` a `$true`.

Bitové operátory nezmiňuji, protože PowerShell nemá podporu pro bitové posuny. Z tohoto pohledu je práce na úrovni bitů nedotažená a nemá smysl se jí zabývat. V případě potřeby se dají najít řešení.

Přiřazení

Většina operátorů přiřazení vychází z aritmetických, tj. `+=` `-=` `*=` `/=` `%=`. A ten nejdůležitější je samozřejmě reprezentován rovnítkem: `=`. Zde není důvod se více rozepisovat. Uvedu jen jeden tip, který nebývá v ostatních jazycích běžný:

```

$a = 1
$b = 2

$b, $a = $a, $b

```

Prohození hodnot dvou proměnných je velmi jednoduché. Ve skutečnosti jde o speciální případ obecnějšího konstruktů – dělení pole:

```

$array = 1,2,3,4
$first, $rest = $array

# můžeme za jistých okolností použít pro iteraci polem
$array = 'jablko',1,'hruska',2,'rybiz',3

```

```
do {
    $ovoce, $index, $array = $array
    Write-Host Ovoce $ovoce ma index $index
} while ($array)
```

Porovnání

Porovnávací operátory jsou jedním z důvodů, proč je PowerShell na první pohled nečitelný. Nešlo o záměr, ale o racionální rozhodnutí. PowerShell byl primárně zaměřen na administrátory a kvůli tomu byl operátor `>` rezervován pro přesměrování do souboru. A od toho se pak už odvinuly další operátory. O jaké se jedná? `-lt -le -eq -ne -ge -gt`. Neboli *less than, less or equal, equal, not equal, greater or equal, greater than*.

Pokud tyto operátory budeme používat na řetězce, je potřeba mít na mysli, že porovnání *neberou* do úvahy case sensitivitu. Pokud potřebujeme přesné porovnání, použijeme operátory `-clt -cle -ceq -cne -cge -cgt`.

Opět připomínám mocné konverze prováděné na pozadí, tedy:

```
1 -eq 01      # žádné přetypování, 01 je 1
1 -eq "1"     # přetypování ze stringu na int
1 -eq "01"    # přetypování ze stringu na int
"1" -eq 01    # přetypování z intu (tj. z 1) na string
"01" -eq 01   # přetypování z intu (tj. z 1) na string ("1")
```

Řetězcové operátory

Pro porovnávání řetězců se naštěstí nemusíme uchýlovat k prostředkům .NET frameworku, nabízí je totiž samotný jazyk: `-match -like -notmatch -notlike`. Rozdíl mezi `-match` a `-like` je možná již na první pohled patrný.

- `-match` bere jako pravý operand regulární výraz a vrací `$true`, pokud levý operand odpovídá regulárnímu výrazu. Jinak vrací `$false`. V prvním případě pak ještě naplní automatickou proměnnou `$matches`, která obsahuje grupy regulárního výrazu.
- `-like` pracuje s wildcardy

```
'123-45-67' -match '\d+- (?<n> (?<n1>\d+) - (?<n2>\d+))' #true
$matches
# dostaneme toto:
#Name      Value
#-----
#n1        45
#n         45-67
#n2        67
#0         123-45-67

'123' -like '1'      #false
'123' -like '1*3'    #true
```

K dispozici máme ještě další operátory. Ty už se používají na manipulaci s textem:

- `-replace` nahrazuje kus textu jiným na základě regulárního výrazu
- `-split` rozděluje text na základě regulárního výrazu
- `-join` spojí objekty do jednoho řetězce

Při použití `replace` můžeme jako nahrazující výraz použít jednoduchý řetězec, ale můžeme se i odkázat na nepojmenovanou groupu (viz. `$0`), nebo pojmenovanou groupu (`${grp}`).

```
'toto je testovací uryvek.' -replace 'to(?:\b)', '$0X'  
'použití groupy' -replace 'ou(<grp>.)', '${grp}${grp}'  
'toto je test' -split '\s+'  
1,2,3,4 -join "+"
```

Operátory na polích (kolekcích)

Doposud jsme používali operátory pro jednoduché typy (`[int]`, `[string]`). Operátory se dají ovšem použít i na kolekce. Zde se ale změní sémantika.

Uvažujme porovnávací operátory. Výsledkem není výraz typu `[bool]`, ale kolekce prvků, které vyhovují podmínce:

```
1..10 + 1..5 -eq 5  
1..10 -lt 5
```

Při použití aritmetických operátorů nedojde k aplikování na prvky kolekce. Zafungují pouze násobení (stejně jako u řetězců) a sčítání (sečte dvě kolekce):

```
1..10 / 3 # chyba  
1..10 + 15 # k poli připojí 15  
1..10 * 2 # pole zduplikuje
```

Operátory typické pro pole pak jsou `-contains` a `-notcontains`. Jejich význam je jasný:

```
1,2,'3' -contains 3  
1,2,'3' -notcontains 3
```

Řetězcové operátory `-match` a `-like` vrátí pouze ty prvky, pro které je vyhodnocení `$true`:

```
'123', '145', '234', '345' -match '^1' # vrátí jen ty, které začínají  
jedničkou  
'123', '145', '234', '345' -like '1*' # obdoba
```

A manipulační řetězcové operátory můžeme použít také:

```
123, '123', '112233' -replace '1', 'X'  
'abc', 'aabbcc', 'abbcc' -split 'b'
```

Všimněte si, že v příkladu na `-replace` jsem jako první položku v poli použil číslo. Připomínám, že opět dojde ke konverzi na řetězec.

Funkce

Deklarace

Funkce musí být uvozena klíčovým slovem *function*. Poté následuje jméno. Parametry a tělo funkce pak můžeme zapsat více způsoby:

```
function mul($what, [int]$times) {  
    $what * $times
```

```

}
function mul {
    param($what, [int]$times)
    $what * $times
}

```

Častěji se používá druhý způsob, který se syntaxí odkazuje na scriptblock a v podstatě jej jen pojmenovává.

Funkci můžeme také vytvořit pomocí cmdletu `New-Item`, v praxi ovšem většinou není důvod ji využít:

```

New-Item -path function: -name mul -value {param($what, [int]$times) $what
* $times}

```

Jako hodnotu zde předáváme parametrizovaný scriptblock. Uvedený příklad využívá toho, že na funkce stejně jako například na souborový systém se můžeme dívat jako na hierarchickou strukturu – drive. Jejich seznam získáme pomocí cmdletu `Get-PsDrive`.

Výše jsme si uvedli, že se scriptblocky (tedy anonymními funkcemi) se dá pracovat jako s jakýmkoliv jiným typem. To stejné platí i pro funkce. Pokud bychom funkci chtěli získat, respektive odkaz na ni, použijeme cmdlet `Get-Item`:

```

function writetest {
    param($prefix) Write-Host $prefix : test -fore Green
}
function writetest2 {
    param($prefix) Write-Host $prefix : test2 -fore Blue
}
function writerCaller {
    # použití operátoru & na zavolání funkce; na konci předáváme parametry
    param($func) & $func 'this is writerCaller'
}
writerCaller (Get-Item function:writetest)
writerCaller (Get-Item function:writetest2)

```

Volání

Právě jsme se dostali k nejdůležitějšímu místu dnešního článku. **Funkce (potažmo cmdlety) voláme s parametry oddělenými mezerou a neuzavřené do závorek.** Toto je jeden z nejčastějších omylů mezi zrychlenými programátory.

```

function mul {
    param($what, $times)
    Write-Host "Multiplying $what * $times"
    $what * $times
}

mul('test', 5)    # ne! na $what se navázalo pole s dvěma prvky
mul ('test', 5)   # ne! na $what se navázalo pole s dvěma prvky
mul 'test', 5     # ne! na $what se navázalo pole s dvěma prvky
mul 'test' 5      # ano
mul test 5        # také možné; zjednodušeně řečeno se neznámá hodnota
považuje za řetězec

```

Výše je uvedeno volání, které spoléhá na pozicování parametrů. Stejně tak ale můžeme specifikovat, na který parametr se argument naváže:

```
mul -times 10 -what 'test'
```

Parametry mohou mít default hodnotu a při volání ani nemusíme všem parametrům argument předat.

```
function mul {
    param($what='test', $times=2)
    $what * $times
}
mul
mul x
mul -times 1
```

V PowerShell se nedají funkce přetěžovat. Místo toho se používá koncept parametrů sdružených do tzv. *parameter set-u*.

```
function Get-Size {
    param(
        [Parameter(ParameterSetName='dir')]$dir,
        [Parameter(ParameterSetName='file')]$file
    )
    if ($PsCmdlet.ParameterSetName -eq 'dir') {
        (Get-ChildItem $dir -recurse |
            ? { !$_.PsIsContainer } |
            Select -expand Length |
            Measure-Object -sum).Sum
    } else {
        Get-Item $file | Select -expand Length
    }
}
```

Více informací může poskytnout přímo PowerShell: `help about_functions_advanced_parameters`.

Pipeline

Mezi nejvýraznější prvky PowerShellu patří pipeline. Jde o vlastnost jazyka, která se používá na iterování prvky kolekce. Krom toho jazyk obsahuje i další konstrukty jako *for*, *while*, *do/while*. Více lze najít v dokumentaci, nebo na internetu. Protentokrát se zaměříme na pipeline.

V oblasti .NET frameworku podporuje pipeline snad jen F#, u ostatních jazyků se rozšíření neplánuje. V F# je pipeline jen syntaktickým cukrem:

```
let col = [1; 2; 5; 10]
col |> List.map (fun i -> printfn "map1: %d" i; i*i)
    |> List.map (fun i -> printfn "map2: %d" i; i*i)
    |> List.iter (printfn "iter: %d")

// ekvivalent:
(List.iter
    (printfn "i3: %d")
```

```

(List.map (fun i -> printfn "i2: %d" i; i*i)
 (List.map (fun i -> printfn "i1: %d" i; i*i) col)))

// výstup:
map1: 1
map1: 2
map1: 5
map1: 10
map2: 1
map2: 4
map2: 25
map2: 100
iter: 1
iter: 16
iter: 625
iter: 10000

```

Zápis s pomocí pipeline je jistě přehlednější. Je vidět, že se jednotlivé řádky a volání funkcí `List.map/iter` volají postupně na všech členech kolekce. V případě použití sekvencí se ovšem výstup radikálně změní:

```

let col = seq { for i in [1;2;5;10] do printfn "yield: %d" i; yield i }
col |> Seq.map (fun i -> printfn "map1: %d" i; i*i)
    |> Seq.map (fun i -> printfn "map2: %d" i; i*i)
    |> Seq.iter (printfn "iter3: %d")

// výsledek
yield: 1
map1: 1
map2: 1
iter3: 1
yield: 2
map1: 2
map2: 4
iter3: 16
yield: 5
map1: 5
map2: 25
iter3: 625
yield: 10
map1: 10
map2: 100
iter3: 10000

```

Zde je vidět důležitý rozdíl mezi oběma výsledky. U sekvencí dochází k *lazy* vyhodnocování, tj. vyhodnocují se až při požadavku (viz. `yield`). Proto se nejdříve zpracuje první prvek, prožene se přes všechna volání `map/iter` a až poté se pokračuje na další prvek. A proč to tu vlastně píšu? Protože PowerShell funguje podobně jako sekvence. Uvedeme si příklad.

```

# vyfiltruje podané objekty a vrátí jen soubory starší než 7 dnů
function FilterFile {
    param([Parameter(ValueFromPipeline=$true)]$item)
    begin {
        Write-Host end: filtering -fore Green
    }
    process {
        if ($item.PsIsContainer) { return } # directory
        if ($item.LastWriteTime -lt (Get-Date).AddDays(-7)) {
            Write-Host Returning $item.Fullname -fore Blue
        }
    }
}

```

```

        $item
    }
}
end {
    Write-Host beg: filtering -fore Red
}
}

# pro každý podaný soubor vrátí jeho první a poslední řádek
function ReturnInterestingLines {
    param([Parameter(ValueFromPipeline=$true)]$file)
    begin {
        Write-Host beg: return interesting -fore Green
    }
    process {
        Write-Host Reading $file -fore Blue
        (Get-Content $file.FullName)[0,-1]
    }
    end {
        Write-Host end: return interesting -fore Red
    }
}

Get-ChildItem $psHome *.txt |
    FilterFile |
    ReturnInterestingLines

```

Zkuste si tento kus kódu zkopírovat do PowerShell konzole nebo ISE a spustit.

- Na výstupu nejdříve vidíme zelený text z *begin* bloků
- Následují modré zprávy o zpracování prvního souboru. Tyto zprávy pochází nejdříve z funkce *FilterFile* a hned poté z *ReturnInterestingLines*. Na výstup jsou pak vypsány příslušné řádky ze souboru. Tady je patrná analogie se sekvencemi v F#. Také se nejdříve zpracuje první položka a až poté následují další.
- Na konci se nachází zprávy o *end* bloku.

OOP

O PowerShellu jistě každý uslyší ve spojení s objekty. V tomto smyslu se o objektově orientované programování jedná. Ale pokud programátor hledá v PowerShellu možnost, jak vytvářet třídu, z ní podědit, mít některé metody virtuální a podobně, bude zklamán.

PowerShell se na tento typ programování nehodí.

Na codeplexu sice existuje [projekt psclass](#), který *implements Inheritance, Polymorphism, encapsulation, and more!*, případně je možné využít cmdletu *Add-Type* a opravdu si vytvořit svou třídu a zkompilevat ji. Přesto bych doporučil odprostit se od naučených objektových technik a využívat typový systém .NET frameworku.

Generiky

V první verzi PowerShellu jsme se museli bez generik obejít. Ve V2 byla dodána podpora, ale bohužel ne kompletní. Můžeme sice vytvořit například `List<string>`, ale už nebyla dodána podpora pro volání generických metod. Čemu to vadí? Pokud jsme dostali neznámou assembly a chceme ji prozkoumat a vyzkoušet si práci s objekty této assembly, pak nemožnost zavolat generickou metodu omezuje naše možnosti.

```
$list = New-Object System.Collections.Generic.List[string]
$list.Add('i1')
$list.AddRange([string[]]('i2', 'i3'))
```

Upozorním jen na dvě volání, která s generikami tolik nesouvisí, ale je dobré si je uvědomit:

```
$list.AddRange('i2', 'i3')      # chyba
$list.AddRange(('i2', 'i3'))    # chyba
```

Při prvním volání by se mohlo zdát, že je všechno v pořádku. PowerShell zde ale neinterpretuje čárku jako operátor. Místo toho zde čárka odděluje jednotlivé argumenty. Druhé volání vyhodí chybu, protože jako argument předáváme *[object[]]*. Automatická konverze zde evidentně neproběhne.

Metody

Jak jsme si už řekli, volání generických metod v PowerShellu nemá přímou podporu. Přesto je ale možné, pokud využijeme prostředků .NET frameworku. Ukážeme si, jak použít již hotové [řešení](#).

```
Add-Type -TypeDefinition @"
    public class TestGenerics {
        public string GetObjectType(T item) {
            return item.ToString() + " - " + typeof(T).FullName;
        }
    }
"@
$t = New-Object testgenerics
d:\Invoke-GenericMethod.ps1 `
    -instance $t `
    -methodName GetObjectType `
    -typeParameters string `
    -methodParameters 'test'
```

Vytvořili jsme si třídu *TestGenerics* a poté ji instancovali a uložili do proměnné *\$t*. Tato třída má jen jednu metodu s generickým parametrem.

Poslední příkaz ukazuje, jakým způsobem je možné generickou metodu zavolat. Využijeme k tomu výše referencovaný skript, který jsme si uložili do souboru *Invoke-GenericMethod.ps1*.

Scope, platnost proměnných

V PowerShellu rozeznáváme tři rozsahy platnosti proměnných: *global*, *script*, *local*.

- *global* označuje scope, který je přístupný pro všechny funkce/cmdlety a vždy je poněkud riskantní v něm něco měnit. Nikdy nevíme, jestli jiný kus kódu neovlivníme.
- *script* je obdoba globálního, ale tentokrát už platí pro jednotlivé skripty. Je vytvořen, když běží skript. Stejně platí i pro modul a uvnitř definované funkce.
- *local* se explicitně nedeklaruje. Je to ale scope, ve kterém jste právě teď. Pokud vytvoříte proměnnou, pak ji vytváříte v *local scope*.

Jednotlivé scope se do sebe zanořují s každým odložením na zásobník. Můžeme se podívat na příklad:

```
function writevar {
    param($func)
    Write-Host $func :: var is $var
}
function test1 {
    $var = 'test1'           # (5)
    writevar test1           # (6)
}
function test2 {
    writevar test2           # (1)
    $var = 'test2'           # (2)
    writevar test2           # (3)
    test1                    # (4), (7)
    writevar test2           # (8)
}
test2
```

Zkusíme si rozebrat volání a vyhodnocování proměnné. Nadeklarovali jsme tři funkce a jdeme volat funkci `test2`. Pokud jsme deklaraci prováděli v konzoli, byli jsme ve scope *global*. V jejím těle se už nacházíme v zanořeném scope, ale *global* máme stále k dispozici.

(1) Jako první krok volá `writevar`. Zanoříme se o jeden scope níž. Při vyhodnocování `$var` se postupuje od aktuálního (*local*) scope směrem nahoru. Nikde ale zatím nebyla `$var` definovaná, takže se vypíše prázdná hodnota.

(2) Dalším krokem je přiřazení `$var = 'test2'`. Toto způsobí, že v lokálním scope se vytvoří proměnná a nastaví jí nějaká hodnota. Tato proměnná bude viditelná i v dalších podřízených scope.

(3) Opět se volá `writevar`. Vytvoří se zanořený scope. Při vyhodnocování se zjistí, že proměnná v lokálním scope neexistuje a jde se o jedno výš (tj. scope funkce `test2`). Zde už byla proměnná definovaná, tedy je při vyhodnocování vrácena její hodnota.

(4) Po vrácení se zpět a zavolání funkce `test1` nastane zajímavá situace: vytvoří se opět zanořený scope v `test1` a v něm se přiřadí do proměnné jiná hodnota (5): `$var = 'test1'`. V tuto chvíli na stacku existují dvě proměnné: s hodnotou `test2` a `test1`. Přiřazením v novém scope tedy nejsme schopni přepsat hodnotu proměnné v nadřazeném scope, místo toho nová proměnná překryje tu dřívější.

Jde o *dynamické vyhodnocování*.

Ve skutečnosti můžeme použít cmdlet `Set-Variable`, který nám proměnnou nastaví v libovolném scope:

```
function inner {
    Write-Host "inner: before" $inouter
    Set-Variable inouter 10 -scope 1
    Write-Host "inner: after" $inouter
}
function outer {
    $inouter = -5
    Write-Host "outer: before" $inouter
    inner
    Write-Host "outer: after" $inouter
}
outer
```

Samozřejmě ale nedoporučuji pro reálné použití.

(6) Voláme z `test1` funkci `writevar`, která si opět vytvoří nový scope. V něm při vyhodnocování proměnnou `$var` nenajde. Pokračuje o jedno výš a prohledává nadřazený scope – to je ten, kde se do proměnné přiřadila hodnota `test1`. Najde a vypíše.

(7) Opustí se funkce `writevar` a hned potom i `test1`. Při tom se zruší scope, kde je definovaná proměnná s hodnotou `test1`.

(8) Proto další vypisování proměnné narazí v nadřazeném scope už jen na proměnnou s hodnotou `test2`.

Proměnná se dá i definovat v *private* scope. Takto definovaná proměnná není viditelná z podřízených scope. V praxi je samozřejmě na překážku vypisovat vždy, že jde o *private*. Proto se běžně používá defaultní *local*, u něhož se modifikátor vynechává.

```
function inner {
    Write-Host inner: $inouter # proměnná není vidět
}
function outer {
    $private:inouter = 10
    inner # nedostane se na proměnnou inouter
}
outer
```

Vyhodnocování výrazů

Výrazem zde rozumím jazykový konstrukt, který vrací nějakou hodnotu. Může to být volání funkce, operátoru na svých operandech a podobně.

```
$d = Get-Date
```

V případě, že PowerShell rozezná, že do nějaké proměnné přiřazujeme hodnotu, pak považuje pravou stranu od rovnítka za výraz, který má vyhodnotit. Proto v proměnné `$d` bude datum a ne reference na funkci. Odlišná situace ale nastává, pokud funkci použijeme jako argument při volání jiného příkazu:

```
Write-Host Get-Date # vypíše get-date
#vs.
Write-Host (Get-Date) # vypíše skutečný datum
```

Zde PowerShell neví, že jde o funkci a `Get-Date` považuje pouze za řetězec (ani ne odkaz na funkci). Proto musíme pomoci *použitím závorek* a vynutit si vyhodnocení.

Při podobných vyhodnocování interpreter postupuje podle přesně definovaných pravidel, kdy se případný výraz snaží vyhodnotit. Některá z nich si ukážeme na příkladech.

```
function testfunc($p) {
    Write-Host Parameter type: $p.GetType() value: $p
}
testfunc get-date # string / get-date
```

```

testfunc (get-date)           # date / aktuální čas
testfunc 1                    # int / 1
testfunc 1.5                  # double / 1.5
testfunc test                 # string / test (1)
testfunc 'test'               # string / test (2)
testfunc ('test')             # string / test (3)
testfunc (test)               # chyba

$d = Get-Item $env:temp       # temporary dir
testfunc $d                   # DirectoryInfo / ..temp
testfunc $d.Name               # string / temp
testfunc $d.FullName.Length   # int / 32 (délka celé cesty)

$f = { Get-Date }             # anonymní fce
testfunc $f                    # scriptblock / {get-date}
testfunc & $f                  # chyba, takto nejde
testfunc (& $f)               # date / aktuální čas

```

Ve funkci `testfunc` jsme neurčili typ parametru `$p`, proto se neprovádí žádné konverze a na parametr se naváže předaná hodnota bez konverze. Proto si bez obav můžeme vypsát typ a hodnotu parametru.

Při předávání řetězce jako argumentu je dobré si zapamatovat, že uvozovky/apostrofy psát nemusíme, proto (1), (2) a (3) jsou ekvivalentní.

Stručně řečeno platí: pokud PowerShell pozná, že daná navazovaná hodnota může reprezentovat výraz, pak tento výraz vyhodnotí. Pokud ne, pak předpokládá, že daná hodnota je řetězec a zachází s ním dále jako s řetězcem.

Hranice hodnoty, kterou se interpreter snaží vyhodnotit jako výraz, je dána mezerami. Proto je už snad jasné, proč následující příkaz skončí chybou:

```
Get-ChildItem C:\Program Files
```

V případě, že si nejsme jistí, jak byly hodnoty navázány na parametry, můžeme použít cmdlet `Trace-Command`:

```
Trace-Command -pshost -name ParameterBinding { Get-ChildItem c:\Program Files }
```

Z výpisu je pak jasné, že na parametr `-Path` byla navázána hodnota `c:\Program` a na parametr `-Filter` hodnota `Files`. Příkaz následně selhal, protože adresář `c:\Program` prostě neexistuje. Řešením je samozřejmě uzavřít cestu do uvozovek.

Výrazy můžeme vytvořit i z řídicích bloků, pokud je zabalíme do `{ ... }`. To se někdy může hodit, pokud potřebujeme vytvořit *one-liner* například kvůli spuštění z `cmd.exe`.

```

# vypíše text modře odpoledne a zeleně jinak
Write-Host modra? zelena? -fore $(if((Get-Date).Hour -ge
12){'Blue'}else{'Green'})

```

A tím jsme se pomalu dostali k vyhodnocování výrazů v řetězcích. Dopředu prozradím, že problematické situace řeší právě blok uzavřený do `{ ... }`. Příklady:

```

write-host "get-date"           # get-date
write-host "(get-date)"        # (get-date)
write-host "$(get-date)"       # aktuální datum

$d = Get-Item $env:temp         # temporary dir
write-host "$d"                # (1) C:\Users\stej\AppData\Local\Temp
write-host "$d.Name"           # (2) C:\Users\stej\AppData\Local\Temp.Name
write-host "$(d.Name)"         # chyba
write-host "$($d.Name)"        # (3) Temp

write-host "$(1; get-date; 'a','b'|%{$_*3})" # (4)

```

Je důležité se zapamovat, že při přístupu k property pomocí tečkové notace se nám vyhodnotí jen samotný objekt (proběhne konverze na string) a část od tečky dál se považuje za řetězec. Poslední příklad (4) navíc ukazuje, že výraz v závorkách může vracet více hodnot. Všechny tyto hodnoty se pak spojí do jednoho stringu.

Co přesně znamená "spojí"? Výraz `$(1; get-date; 'a','b'|%{$_*3})` vrátí pole prvků. První bude číslo, druhý bude datum a poté dva řetězce. Toto pole je poté konvertováno na řetězec. Při konverzi je použita speciální proměnná `$ofs`, jejíž výchozí hodnota je mezera. Zkusme ji změnit na cokoli jiného a uvidíme výsledek:

```

$ofs = ']['
write-host "$(1; get-date; 'a','b'|%{$_*3})"

```

Závěrem

Snažil jsem se pohlédnout na PowerShell očima programátora a vypíchnout ty vlastnosti, které mohou být pro programátora zajímavé a kde může narazit na rozdíly mezi svým jazykem a PowerShellem. Článek tentokrát narostl do délky, přesto věřím, že se najde někdo, kdo ho přečte celý. V příštím díle se už zkusíme podívat na praktičtější použití.

REGULÁRNÍ VÝRAZY A POWERSHELL

V [minulém díle](#) jsme si ukázali prohledávání textu pomocí operátorů a cmdletu **Select-String**. Dnes hledání rozšíříme o regulární výrazy.

Some people, when confronted with a problem, think
 “I know, I’ll use regular expressions.” Now they have two problems.

Tento známý citát (více viz [Source of the famous “Now you have two problems” quote](#)) mluví asi za vše. Především panuje jistá averze vůči regulárním výrazům „protože jsou složité a divný“. Nechci se pouštět do ukázek typu „takhle složitě se to dá udělat“, příklady si můžete najít sami např. na [regexlib.com](#).

Operátor –match

V minulém díle jsem schválně přeskočil jeden operátor: **-match**. Tento operátor vyhodnotí regulární výraz oproti danému textu a vrátí hodnotu **\$true**, pokud je nalezena shoda. Výsledek je uložen do proměnné **\$matches**. Ukážeme si na něm některé základní konstrukce pro tvorbu regulárních výrazů. Vždy platí, že na levé straně operátoru **-match** uvádíme text, ve kterém hledáme a na pravé straně zapisujeme daný regulární výraz.

```
PS C:\> 'PowerShell' -match 'hell'
True
PS C:\> $matches
```

Name	Value
----	-----
0	hell

V tomto nejjednodušším případě jsme hledali, jestli je ve vstupním textu „PowerShell“ obsažen text „hell“ (což je v tomto případě regulární výraz skládající se ze čtyř znaků řazených danou posloupností). Operátor vrátil hodnotu **\$true** a proměnná **\$matches** obsahuje část textu, který splňuje zadání.

```
PS C:\> 'PowerShell' -match 'he..'
True
PS C:\> $matches
```

Name	Value
----	-----
0	hell

Tečka zastupuje jakýkoli znak. Je vidět, že ve výsledku jsou tečky nahrazeny znaky *l*.

```
PS C:\> 'PowerSuperShell' -match '(Power).*(Shell)'
True
PS C:\> $matches
```

Name	Value
----	-----
2	Shell
1	Power
0	PowerSuperShell

Trošku přitvrdíme, ale zůstaneme na zemi. Pomocí závorek vytvoříme skupiny, každá ze skupin je poté v proměnné **\$matches** uložena jako jeden záznam.

Dále jsme zavedli hvězdičku, která nám říká, že znak předcházející se může opakovat libovolněkrát (i 0krát!). Přeloženo do češtiny, předchozí regulární výraz znamená: Najdi znaky/slovo *Power*, poté přeskoč libovolný počet znaků a najdi znaky/slovo *Shell*. Ve výsledku vidíme, že na pozici 0 se uložil celý prohledávaný text, na pozici 1 první hledaný text a na pozici 2 druhý hledaný text. Mimo znaku hvězdičky jsou v regulárních výrazech ještě další dva podobné: otazník (?) a znaménko plus (+). Nejprve ukážeme příklad.

```
PS C:\> 'PowerShelllll' -match 'el+'
True
PS C:\> $matches
```

Name	Value
----	-----
0	ellll

```
PS C:\> 'PowerShelllll' -match 'el?'
True
PS C:\> $matches
```

```
Name      Value
-----
0         e
```

První příklad říká, že znak před + se může vyskytnout v textu jednou a vícekrát, proto vidíme ve výsledku znak *l* opravdu čtyřikrát. Naproti tomu druhý příklad – otazník – říká, že znak před ním se může objevit jednou a nebo vůbec! Proto se nám znak *l* ve výsledném textu neobjeví. Fakticky máme ve výsledku již první výskyt písmene *e* (ze slova *Power*), protože splňuje podmínku, že se za ním znak *l* nevyskytuje. Shrňme si předchozí odstavce do jednoduché tabulky.

Znak	Význam
*	Žádný nebo libovolný počet výskytů
?	Žádný nebo jeden výskyt
+	Jeden nebo více výskytů

Zkuste si za pomoci předchozí tabulky určit výsledky následujících ukázek.

```
PS C:\> 'aaa' -match 'a*'
PS C:\> 'aaa' -match 'a?'
PS C:\> 'aaa' -match 'a+'
PS C:\> 'aaa' -match 'b*'
PS C:\> 'aaa' -match 'b?'
PS C:\> 'aaa' -match 'b+'
```

Kolikrát myslíte, že výsledkem bude **\$true**? Pokud jste „tipovali“ že pětkrát, máte pravdu. Pokud jste výsledek netrefili, zkuste se znovu podívat do předchozí tabulky a koukněte se následující výsledky:

```
PS C:\> 'aaa' -match 'a*'; $matches
True
```

```
Name      Value
-----
0         aaa
```

```
PS C:\> 'aaa' -match 'a?'; $matches
True
```

```
Name      Value
-----
0         a
```

```
PS C:\> 'aaa' -match 'a+'; $matches
True
```

Name	Value
----	-----
0	aaa

```
PS C:\> 'aaa' -match 'b*'; $matches
True
```

Name	Value
----	-----
0	

```
PS C:\> 'aaa' -match 'b?'; $matches
True
```

Name	Value
----	-----
0	

```
PS C:\> 'aaa' -match 'b+'; $matches
False
```

Name	Value
----	-----
0	

Ještě jednou upozorňuji na „zradu“ v podobě formulace „žádný výskyt“. Proto i zcela „nesmyslný“ test ('a' -**match** 'b?') vrátí hodnotu **\$true**. Nyní už určitě chápete citát ze začátku článku .

Občas se hodí vědět přesný počet výskytu určitých znaků. V tomto případě lze využít rozsah uvedený ve { složených závorkách }.

```
PS C:\> 'abbabbbbabbbbbb' -match 'ab{2}'; $Matches
True
```

Name	Value
----	-----
0	abb

```
PS C:\> 'abbabbbbabbbbbb' -match 'ab{3,}'; $Matches
True
```

Name	Value
----	-----
0	abbbb

```
PS C:\> 'abbabbbbabbbbbb' -match 'ab{5,8}'; $Matches
True
```

Name	Value
----	-----
0	abbbbbbb

{2} udává, že hledáme přesně dva výskyty znaku před závorkou, {3,} určuje, že chceme tři a více výskytů a konečně {5,8} říká, že hledáme četnost mezi pěti a osmi výskyty. Nyní už se můžeme pustit i do o něco složitějších konstrukcí. Potřebujete zkontrolovat telefonní číslo? Všechny následující příklady vrátí **\$true**, pokud zadáte devítimístné číslo.

```
PS C:\> '123456789' -match '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]'
True
```

```
PS C:\> '123456789' -match '[0-9]{9}'
True
```

```
PS C:\> '123456789' -match '\d{9}'
True
```

Vzpomínáte na operátor rozsahu z minulého dílu? `[0-9]` říká, že hledáme čísla v daném rozsahu. V prvním příkladu tento rozsah zkopírujeme devětkrát, ve druhém použijeme daný počet výskytů `{9}`. Poslední příklad využívá definovanou množinu znaků (character class) - `\d` znamená jakoukoli číslici. Vzhledem k tomu, že se dnešní díl opět trochu natáhl, povíme si o množinách znaků více příště.

Při práci s operátorem **-match** se vám stane, že při použití ve funkci vám začne vadit vypisování **\$true** nebo **\$false** při každém porovnání. Pokud chcete pracovat pouze s výsledky, můžete výstup přesměrovat pomocí cmdletu **Out-Null**. Porovnejte následující výstupy.

```
PS C:\> $p = 'PowerShell'
PS C:\> foreach ($a in $p,$p,$p,$p,$p) { $a -match 'hell' }
True
True
True
True
True
PS C:\> foreach ($a in $p,$p,$p,$p,$p) { $a -match 'hell';
$matches[0] }
True
hell
True
hell
True
hell
True
hell
True
hell
PS C:\> foreach ($a in $p,$p,$p,$p,$p) { $a -match 'hell' | Out-
Null; $matches[0] }
```

hell
hell
hell
hell
hell

V prvním příkladu vidíme pouze logický výsledek regulárního výrazu. Ve druhém přidáváme i výsledek, pokud chceme vidět pouze tento výsledek, použijeme zmiňovaný cmdlet **Out-Null**.

NOVINKY POWERSHELL VERZE 3

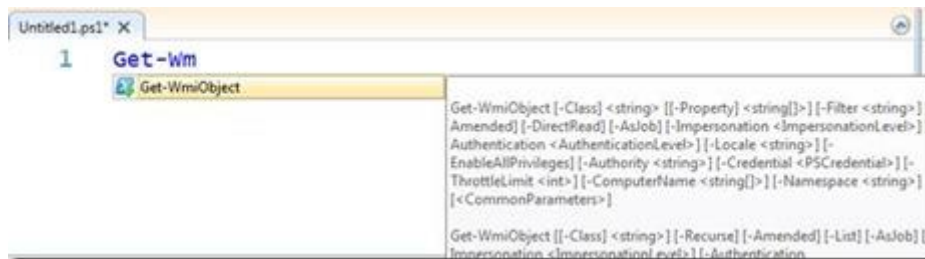
Dnes se nebudeme věnovat plánovanému tématu (regulární výrazy), ale celý článek bude o aktualitách posledního měsíce.

PowerShell v3

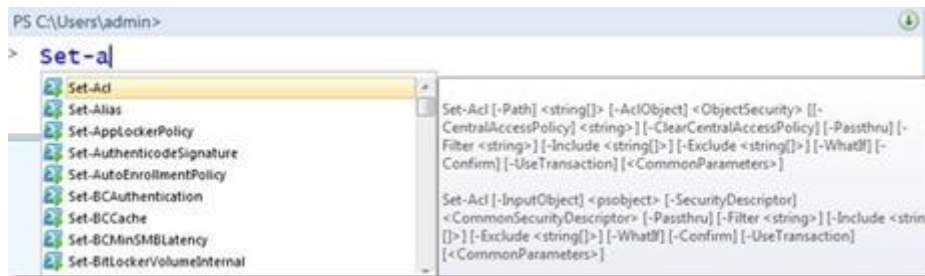
Možná vám neuniklo, že světlo světa spatřil PowerShell v3 – tradááá Zatím pouze jako [CTP1](#) (Community Technology Preview), ale i tak se jedná o událost hodnou zaznamenání. Už nyní jsou vidět některé velice zajímavé novinky. Pojdme se podívat na ty, které jsou dle mého názoru nejzajímavější.

Nejviditelnější změny proběhly v grafickém editoru PowerShell ISE. Jedná se především o vylepšený IntelliSense, který se tím dostal na úroveň editorů třetích stran (např. PowerGUI, PowerSE,...). Více v následujících obrázcích:

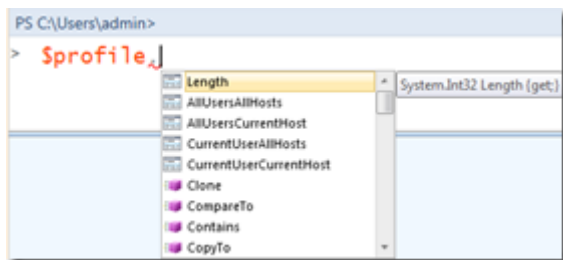
IntelliSense – jak mám rád tuhle „novinku“. Základní věc, podle které posuzuji editory. Konečně jsme se jí dočkali i v ISE.



Naštěstí funguje i v panelu příkazového řádku:



IntelliSense zobrazuje (stejně jako například Visual Studio) různé ikony pro metody a vlastnosti objektů:



Pro mne rozhodně jedna z nejlepších vizuálních novinek. Už i z toho důvodu, že nyní nebude potřeba instalovat na servery editory třetích stran.

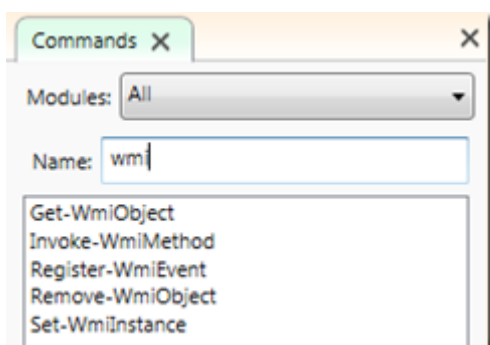
Další zajímavou novinkou v ISE je Add-on nazvaný **Commands**. Tento Add-on zjednodušuje (hlavně novým uživatelům PowerShellu) vyhledávání potřebných cmdletů a jejich parametrů. Pokud víte, že chcete pracovat s WMI, můžete použít buď Get-Command

```
PS C:\Scripts > gcm *wmi* -c cmdlet
```

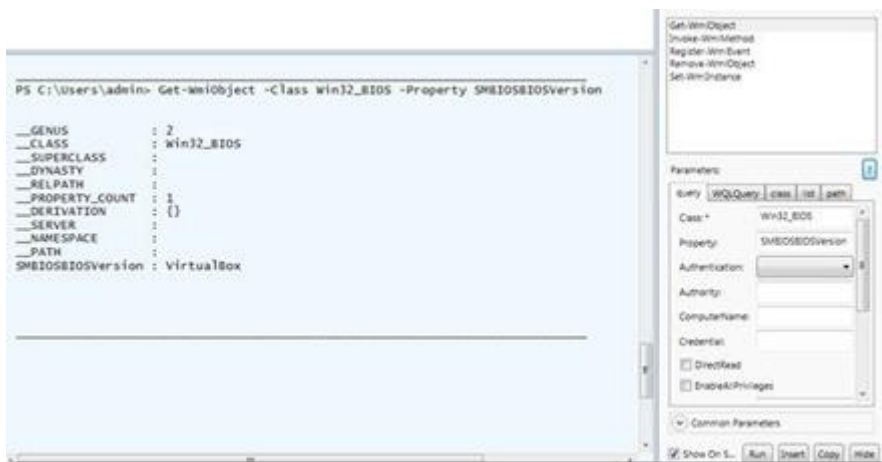
CommandType Name Definition

```
-----
Cmdlet Get-WmiObject Get-WmiObject [-Class] <Stri...
Cmdlet Invoke-WmiMethod Invoke-WmiMethod [-Class] <S...
Cmdlet Register-WmiEvent Register-WmiEvent [-Class] <...
Cmdlet Remove-WmiObject Remove-WmiObject [-Class] <S...
Cmdlet Set-WmiInstance Set-WmiInstance [-Class] <St...
```

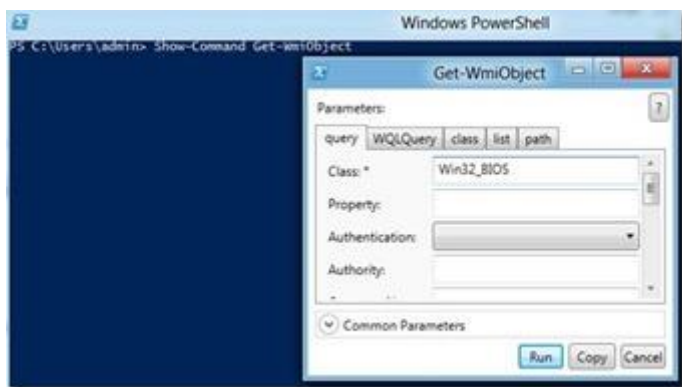
Nebo přímo zmiňovaný Add-on:



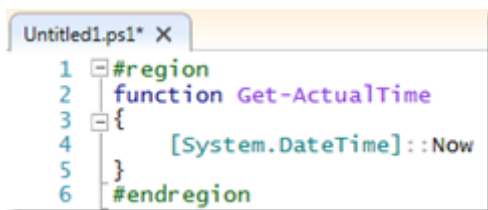
Pro vybraný cmdlet, můžete okamžitě vyplnit potřebné parametry a spustit jej:



Tento Add-on můžete spustit (i z PowerShell konzole) pomocí cmdletu **Show-Command**.



ISE podporuje nově také regiony:



Abych nezapomněl – ISE (chce se mi napsat – konečně) umí zobrazit seznam posledních otevřených souborů.

Zajímavé změny se dočkal cmdlet **Where-Object**, osobně si myslím, že se jedná o dobrý krok směrem ke zpřístupnění začínajícím PowerShellistům:

```
PS C:\Scripts > Get-Process | Where-Object { $_.Name -like 'ls*' }
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
594	10	4828	616	44	153.88	588	lsass

```
PS C:\Scripts > Get-Process | Where-Object Name -like 'ls*'
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
594	10	4828	616	44	153.88	588	lsass

Všimněte si druhého příkladu. Již není potřeba zápisu `$_` pro aktuální objekt v rouře. Nový zápis je čitelnější.

Tak to byly nejviditelnější změny v PowerShellu v3 CTP1. Změn je samozřejmě o mnoho víc. Pokud vás zajímají, zkuste se podívat na některý z následujících článků:

- <http://dmitrysotnikov.wordpress.com/2011/09/14/first-glimpse-at-powershell-v3/>
- <http://www.jonathanmedd.net/2011/09/powershell-v3-a-quick-first-look.html> - v tomto článku je popsána novinka v načítání modulů.
- <http://www.powershellmagazine.com/2011/09/15/how-to-find-out-whats-new-in-powershell-vnext/>
- <http://www.powershellmagazine.com/2011/09/28/powershell-v3-ise-and-ise-scripting-model-changes-improvements/> - novinky v ISE z pohledu \$psISE.

PowerShell Deep Dive

V polovině října se mi splnil velký sen. Dostal jsem se na konferenci pořádanou firmou Quest – The Expert Conference (TEC). Firma Quest stojí za známým editorem PowerGUI a cmdlety pro správu Active Directory. Součástí TEC byla i první evropská konference, jejíž obsah byl věnován pouze PowerShellu – PowerShell Deep Dive. Po několika letech jsem měl možnost potkat naživo množství

PowerShell guru, které jsem do té doby znal pouze z Twitteru nebo různých online meetingů. Na konferenci přijelo i několik členů PowerShell týmu včetně Bruce Payetta, designera jazyka PowerShellu a autora vynikající knihy PowerShell in Action (mimořádně Bruce na můj dotaz, jestli se dočkáme knihy PowerShell in Action v3 odpověděl, že pravděpodobně ne – doufám, že změní názor :). Pojdme si v krátkosti představit ty nejzajímavější přednášky – třeba vás navnadím na Deep

Dive 2012

Dmitry Sotnikov: Get your jobs done! Dmitry mluvil o jobech a ukázal základní cmdlety pro práci ve verzi 2 a také ve verzi 3. Tato přednáška nepřinesla nic nového z hlediska tipů pro práci, ale ukázala výhodu toho, když máte někoho z PowerShell týmu v publiku. Dmitry totiž poukázal na fakt, že při práci ve v3 se občas při zobrazení vlastností jobu (pomocí Get-Job) zobrazí parametr StatusMessage s textem test. Otázku směřoval na Bruce Payetta. Bruce začal cosi tukat do svého notebooku a při své další prezentaci nám ukázal kus zdrojového kódu PowerShellu a vysvětlil, proč se onen text test do výsledku dostal (jednalo se o chybu člověka, který programoval zmiňovaný

cmdlet). Super ukázka rychlé reakce na podněty

Brandon Shell : Notes from the Fields, Production Module Design. Dvě přednášky, které Brandon věnoval povídání o své praxi při vytváření modulů ([splunk](#) a [BSonPoSh](#)) a při zavádění PowerShellu do praxe. Pro další studium Brandon doporučil článek od Tome Tanasovského [Modules – Modules – Modules](#).

Aleksandar Nikolic: Delegated Administration. Alex je jeden z největších expertů na použití vzdálené administrace pomocí PowerShellu. Je mimo jiné i spoluautorem vynikající příručky [Administrators Guide to Windows PowerShell Remoting](#). V této přednášce ukazoval možnosti nastavení tzv. Fan-in scénáře za použití delegování rolí.

Shay Levy & Kirk Munro: Proxy functions. Pro mne jednoznačně nejlepší přednáška celé konference. Shay a Kirk ukazovali možnosti při vytváření proxy funkcí. Nejprve Shay ukázal jak na proxy funkce pomocí standardních technik (můžete se podívat například na jeho starší článek [Proxy Functions: Spice Up Your PowerShell Core Cmdlets](#)) a poté Kirk předvedl modul na kterém se Shayem pracovali – [PSPX](#) (PowerShell Proxy Extensions). Tento modul opravdu výrazně zjednodušuje vytváření proxy funkcí, proto bych se u něj rád na chvíli zastavil.

Jedna z věcí, která je PowerShell týmu vytýkána, jsou parametry cmdletu Get-ChildItem. Pro vylistování např. pouze adresářů musíte použít cmdlet Where-Object (Get-ChildItem | Where-Object { \$_.PSIsContainer }). Ideální by bylo, kdyby cmdlet Get-ChildItem rovnou obsahoval možnost zobrazit adresáře (mimořádně ve v3 je již toto vyřešeno). Ve v2 musíte použít proxy funkce. Díky modulu PSPX stačí zapsat pouze:

```
PS Dropbox:\PowerShell\Scripts>
> Fix-It Get-ChildItem -Parameter @{
    Name = 'Directory'
    PostProcess = {
        Where-Object {$_.PSIsContainer}
    }
} -DefineNow
```

Poté již můžete použít příkaz

```
PS C:\> Get-ChildItem -Directory
```

A cmdlet vám vrátí pouze adresáře. Rozhodně vám doporučuji uvedený modul prozkoumat.

Bruce Payette měl dvě přednášky – **PowerShell Workflow** a **Concurrent Operations**. Workflows jsou novinkou ve v3 a Bruce ukazoval jejich základní použití. Druhá přednáška byla pro mne trochu jako z jiného světa a rozhodně si počkám až bude uvolněn záznam, abych si mohl celou přednášku projít hezky v klidu ještě jednou.

James O'Neill: Maximize the Reuse of your PowerShell. Velmi zajímavá přednáška na téma moduly. James je autorem modulu pro Hyper-V a má množství zkušeností z praxe. Výťah z jeho [přednášky](#) si můžete přečíst na jeho blogu, kde je uveden i link na slidy z Frankfurtu.

Dalším, kdo zveřejnil slidy a kód z přednášky je i Jeffery Hicks, který měl téma [Turn Command Line Tools into PowerShell Tools](#).

Lightning Rounds – velmi zajímavý koncept krátkých pětiminutových přednášek. Bylo jich celkem deset a zúčastnit se mohli i lidé z „publika“.

Script Club – každý večer po skončení oficiálního programu jsme se ještě téměř všichni sešli, abychom probrali témata, která nás zajímala. Toto byla vynikající příležitost popovídat si s členy PowerShell týmu, MVP či ostatními a rozebrat vlastní skripty. Poslední den nás musela z konferenční místnosti vyhodit uklízející parta a i potom někteří pokračovali v diskusi v hotelovém baru.

Mimo oficiální program proběhly ještě dvě krátké diskuse. Jedna na téma PowerGUI a druhá na téma Quest AD cmdlets. V průběhu obou jsme se dozvěděli, co nového lze čekat v následujících verzích produktů a také jsme mohli přímo s členy vývojových týmů probrat, co nás zajímá a co bychom případně chtěli v těchto verzích mít.

Rozhodně můžu říct, že mě osobně konference dala opravdu hodně. Upřímně řečeno – obsah bych klidně snesl o něco techničtější, ale možnost vidět po letech lidi, které jsem znal pouze z virtuálního světa byla opravdu k nezaplacení. Stejně tak jako volné konverzace mezi přednáškami či mimo ně. Jestli to půjde, příští rok pojedu znovu. Pokud se chcete nasát atmosféru z Deep Dive, můžete se podívat na mé [fotky](#) z celé podařené akce.

A dnes již definitivně poslední zpráva. Začal vycházet PowerShell Magazine. Na adrese <http://www.powershellmagazine.com/> se můžete dočíst množství zajímavých informací a novinek z první ruky. Jména, která za vznikem stála jsou dle mého dobrou zárukou kvalitního obsahu.

PŘEDSTAVENÍ PSCX

V dnešním povídání o PowerShellu představím modul [PSCX](#), neboli PowerShell Community Extensions. Na jeho vývoji se podílí jedni z mála vývojářských PowerShell MVP, proto se dá čekat jiné zaměření, než u většiny PowerShell modulů bývá běžné.

Modul lze stáhnout na adrese <http://pscx.codeplex.com>. V release notes se detailně popisuje, jak modul zprovoznit. Stručně řečeno stačí pouze odblokovat stáhnutý zip soubor (pokud jste stahovali pomocí Internet Exploreru; ostatní prohlížeče informaci o stáhnutí nenastavují), anebo nastavit policy na **Unrestricted** a je hotovo. Obsah zip souboru se pak rozbalí na cestu z proměnné prostředí **\$env:PsModulePath**. Pokud takovou proměnnou prostředí ještě nemáte, je potřeba ji vytvořit. Jednou z rozumných hodnot je do C:\Windows\system32\WindowsPowerShell\v1.0\Modules\. V tomto se postup neliší od "instalace" jakéhokoliv jiného modulu.

Import modulu

Po nainstalování se modul jednoduše naimportuje příkazem **Import-Module pscx** (anebo aliasem **ipmo**). Pokud chcete vidět, které všechny příkazy (cmdlet, funkce, aliasy) byly naimportovány, pomůže přepínač **-verbose**, případně lze později příkazy získat pomocí **Get-Command**:

```
PS> ipmo pscx -Verbose -Force

VERBOSE: Loading module from path 'C:\Users\stej\Documents\WindowsPowerShell\Modules\pscx\pscx.ps...'

VERBOSE: Loading 'Assembly' from path 'C:\Users\stej\Documents\WindowsPowerShell\Modules\pscx\Psc...'

VERBOSE: Loading 'TypesToProcess' from path 'C:\Users\stej\Documents\WindowsPowerShell\Modules\pscx\TypesToProcess.ps...'

...
```

```

VERBOSE: Loading module from path 'C:\Users\stej\Documents\WindowsPowerShell\Modules\pscx\Pscx.dll'.
VERBOSE: Importing cmdlet 'Out-Clipboard'.
VERBOSE: Importing cmdlet 'Expand-Archive'.
...
VERBOSE: Importing function 'Show-Tree'.
VERBOSE: Importing function 'Stop-RemoteProcess'.
VERBOSE: Importing alias '?:'.
VERBOSE: Importing alias '??'.
VERBOSE: Importing alias 'call'.
...

```

```
PS> > Get-Command -module pscx
```

CommandType	Name	Definition
Alias	?:	Invoke-Ternary
Alias	??	Invoke-NullCoalescing
Function	Add-DirectoryLength	...
Cmdlet	Add-PathVariable	Add-PathVariable [-Value] [-Name] [-Prepend] ...
Function	Add-ShortPath	...

Příkazů nainportovaných z PSCX modulu je velké množství. Pro lepší orientaci a představu můžeme zkusit seskupit příkazy podle podstatného jména:

```

PS> > Get-Command -module pscx |
    group -prop { $_.Name -replace '^.*?-','' } |
    sort count -desc

```

Count	Name	Group
6	MSMQQueue SMQueue...	{Clear-MSMQQueue, Get-MSMQQueue, New-MSMQQueue, Receive-M
4	Clipboard	{Get-Clipboard, Out-Clipboard, Set-Clipboard, Write-Clipboard}
3	Xml	{Convert-Xml, Format-Xml, Test-Xml}
3	TerminalSession	{Disconnect-TerminalSession, Get-TerminalSession, Stop-TerminalSession}
3	Bitmap	{Export-Bitmap, Import-Bitmap, Resize-Bitmap}
3	AlternateDataStream	{Get-AlternateDataStream, Remove-AlternateDataStream, Test-AlternateDataStream}

3 PathVariable	{Add-PathVariable, Get-PathVariable, Set-PathVariable}
3 EnvironmentBlock EnvironmentBlock}	{Get-EnvironmentBlock, Pop-EnvironmentBlock, Push-EnvironmentBlock}
2 ForegroundWindow	{Get-ForegroundWindow, Set-ForegroundWindow}
2 File	{Edit-File, Unblock-File}
...	

V tomto krátkém seznámení není prostor na podrobné představení jednotlivých příkazů. Z toho důvodu v následujících odstavcích vyberu pouze některé. Pořád ale platí, že PSCX je modul plný perliček!

Ternární operátor a podobné pomůcky.

Také jste už narazili na konstrukce jako jsou tyto následující?

```
$a = if ($somevar) { 'set' } else { 'unset' }
# anebo
if ($somevar) {
    $a = 'set'
} else {
    $a = 'unset'
}
```

A také jste si vzpomněli na ternární operátor známý z C#? Pak vás bude zajímat funkce `Invoke-Ternary`, nebo ještě lépe alias `?:`. Zápis je kratší a čitelnější.

```
$a = ?: { $somevar } { 'set' } { 'unset' }
$a = ?: { gci d:\temp } { 'something in temp' } { 'temp empty' }
```

"Operátor" ovšem nejde řetězit. Případné další vyhodnocování je tak nutné uzavřít do posledního `else` bloku.

```
?: { gci d:\temp\ } `
{ 'something in temp' } `
{ ?: { gci d:\temp2 } `
{ 'something in temp2' } `
{ 'even temp2 is empty' } }
```

Podobně vývojářský orientovaný je cmdlet `Invoke-NullCoalescing`, ke kterému existuje alias `??`. Podle jména aliasu se dá zřejmě vytušit, k čemu se takový cmdlet dá použít.

```
PS> ?? { $Env:PSModulePath } { "$psHome\modules" }
PS> ?? { $Env:PSModulePath } { throw '$env:PsModulePath does not exist' }
```

Je dobré si uvědomit, že oba příkazy pracují se *scriptblocky*. Šlo by sice definovat příkazy tak, aby například fungovalo volání `?: { gci d:\temp } 'something in temp' 'temp empty'` (tj. poslední

dva parametry jsou předány standardně hodnotou), ale tím by se ztratila flexibilita poskytovaná *scriptblockem*.

Poslední příkaz orientovaný převážně na vývojáře se jmenuje **Invoke-Method** s aliasem **call**. Ten najde uplatnění v okamžiku, kdy pracujeme s kolekcí objektů a na každém z nich potřebujeme zavolat nějakou metodu.

```
PS> $o = [datetime]'2012-01-01', [datetime]'2012-01-02', [datetime]'2012-01-03'
PS> $methods = 'AddHours', 'AddMinutes', 'AddSeconds'
PS> foreach ($m in $methods) {
>> $o | Invoke-Method $m ([double]1.0)
>>}

1. ledna 2012 1:00:00
2. ledna 2012 1:00:00
3. ledna 2012 1:00:00
1. ledna 2012 0:01:00
2. ledna 2012 0:01:00
3. ledna 2012 0:01:00
1. ledna 2012 0:00:01
2. ledna 2012 0:00:01
3. ledna 2012 0:00:01
```

Všimněte si přetypování na **[double]**. Funkce **Invoke-Method** není tak inteligentní, že by konverzi typů řešila za nás. V tomto případě by to možné bylo, ale obecně bychom se mohli dočkat nečekaných překvapení.

Stejného efektu jako u **Invoke-Method** se dá samozřejmě dosáhnout i pomocí **ForEach-Object**, ale už musíme znát jméno metody:

```
$methods = 'AddHours', 'AddMinutes', 'AddSeconds'
PS> foreach ($m in $methods) {
>> $o | % { $_."$m"(1.0) # chyba
>>}

Unexpected token '(' in expression or statement.
...

PS> $o | % { $_.AddHours(1) } # ok
```

Podobný příkaz pro property nebyl potřeba, protože u properties toto funguje samo:

```
PS> foreach ($m in 'year', 'month', 'day') {
>> $o | % { $_."$m" }
```

```
>> }
```

```
2012
```

```
2012
```

```
2012
```

```
1
```

```
1
```

```
1
```

```
1
```

```
2
```

```
3
```

Mezi další vývojářské příkazy se řadí například také `Get-LoremIpsum`, `Invoke-GC` (invoke garbage collector), `Invoke-Reflector` (již mírně neaktuální, že?), `Invoke-Apartment` (spuštění PowerShell kódu v daném apartmentu – STA/MTA).

Formátování

Do této skupiny zahrnu dva příkazy, které čas od času mohou pomoci. První naformátuje xml vstup, druhý pak zobrazí obsah adresáře ve stromové struktuře (nemusí ale jít jen o adresář na disku).

První příkaz `Format-Xml` zobrazí obsah XML souboru pěkně naformátovaný.

```
PS> $xml = [xml]('<root><post id="1" date="sun">first post</post>' +  
  '<post id="2" date="mon">first post</post></root>')  
PS> $xml | Format-Xml -AttributesOnNewLine
```

```
<root>  
  <post  
    id="1"  
    date="sun">first post</post>  
  <post  
    id="2"  
    date="mon">first post</post>  
</root>
```

`Format-Xml` umí zpracovat a naformátovat i soubor, pokud mu jako argument podáme cestu na disk. Druhý příkaz jménem `Show-Tree` bude nejlepší také rovnou ukázat na příkladu.

```
PS> Show-Tree G:\Programs\IrfanView425\  
G:\Programs\IrfanView425\  
|--Html  
|--Languages
```

```

|--Plugins
|   |--Adobe 8BF
|   |--Crw
|   |--Ecw
|   |--Filter Factory 8BF
|   \--Fmod
\--Toolbars

```

Pokud nespecifikujeme jinak, příkaz zobrazí pouze adresáře (obecně kontejnery). Příkaz má samozřejmě sadu přepínačů, z nichž pro nás nejzajímavější budou **-ShowProperty** a **-ShowLeaf**.

```
PS> Show-Tree G:\Programs\IrfanView425\Plugins -ShowLeaf #zobrazí soubory
```

```
G:\Programs\IrfanView425\Plugins
```

```

|--Adobe 8BF
|   |--HarrysFilters.8bf
|   \--PopArt.8bf
|--Crw
|   \--Readme_Canon.txt
|--Ecw
|   |--NCScnet.dll
|   |--NCSEcw.dll
|   |--NCSEcwC.dll
|   \--NCSUtil.dll

```

zobrazí soubory a property k souborům a adresářům

```
PS> Show-Tree G:\Programs\IrfanView425\Plugins\Crw -ShowProperty -ShowLeaf
```

```
G:\Programs\IrfanView425\Plugins\Crw
```

```

|--Property: Attributes = Directory, Archive
|--Property: CreationTime = 11/30/2010 19:04:58
|--Property: LastAccessTime = 11/30/2010 00:00:00
|--Property: LastWriteTime = 11/30/2010 19:05:00
|--Property: Mode = da---
...

```

```
\--Readme_Canon.txt
```

```

    |--Property: Attributes = Archive
    |--Property: CreationTime = 11/30/2010 19:04:58
    |--Property: Directory = G:\Programs\IrfanView425\Plugins\Crw

```

...

Show-Tree tedy názorně zobrazí adresářovou strukturu včetně properties jednotlivých položek. Cesta ovšem může ukazovat i jinak než na disk. Jinak řečeno můžeme použít cestu z jakéhokoli providera. Co třeba udělá **Show-Tree cert: -ShowLeaf**? Anebo

```
PS> Show-Tree 'HKCU:\Software\Microsoft\Internet Explorer\main' -ShowProperty
HKCU:\Software\Microsoft\Internet Explorer\main
|--Property: Anchor Underline = yes
|--Property: AutoHide = yes
|--Property: Cache_Update_Frequency = Once_Per_Session
|--Property: CompatibilityFlags = 0
|--Property: Disable Script Debugger = yes
|--Property: DisableScriptDebuggerIE = yes
...
\--WindowsSearch
    |--Property: Cleared = 1
    |--Property: Cleared_TIMESTAMP = ...
    \--Property: Version = WS not running
```

V tomto okamžiku je jasné, že na rychlé prozkoumání registrů se tento příkaz velmi dobře hodí. Najednou je schopen zobrazit informace o struktuře i o hodnotách.

Jen pro úplnost musím zmínit, že máme k dispozici i hexa formátování díky příkazu **Format-Hex**.

CD

Jeden z užitečných příkazů, který přetěžuje původní funkci, se skrývá za aliasem **CD**. Příkaz provede změnu adresáře, jak by každý čekal. Krom toho si ale vnitřně udržuje i seznam již navštívených adresářů a dá se v tomto seznamu i pohybovat.

```
PS> cd hkcu:\Software\Microsoft\VisualStudio\10.0\FileMRUList
PS> cd HKCU:\Software\7-ZIP
PS> cd d:\temp
PS>$pwd
Path
----
D:\temp

PS> cd

# Directory Stack:
--- -----
```

```
0 C:\prgs\tools\Console2
1 HKCU:\Software\Microsoft\VisualStudio\10.0\FileMRUList
2 HKCU:\Software\7-ZIP
-> 3 D:\temp
```

Prosté vypsání seznamu tedy dosáhneme pomocí `cd` bez argumentů. Pro pohyb zpět použijeme `cd -`, pro pohyb vpřed pak `cd +`. Pokud známe konkrétní číslo, dostaneme se na něj pomocí `cd -cislo`

```
PS> cd -
PS> cd

# Directory Stack:
---
0 C:\prgs\tools\Console2
1 HKCU:\Software\Microsoft\VisualStudio\10.0\FileMRUList
-> 2 HKCU:\Software\7-ZIP
3 D:\temp
PS> cd -0; $pwd # skok na první záznam
Path
----
C:\prgs\tools\Console2
```

Druhou drobnou vychytávkou je posun v adresářové struktuře směrem k rootu, ale o více parentů. O kolik přesně to je, to záleží na počtu teček. Posun o jednoho parenta značí dvě tečky. O dva parenty tři tečky atd.

```
PS> cd HKCU:\Software\Microsoft\VisualStudio\10.0\FileMRUList
PS> cd .....; $pwd
Path
----
HKCU:\Software
```

Blbinka

Nevím, jestli tato funkce má nějaký praktický význam, ale autoři ji do modulu zařadili. Její jméno je `Out-Speech`. Jde o funkci, která předčítá daný text. Jak by tedy mohlo vypadat pročtení Hamleta před spaním?

```
PS> $url = 'http://www.gutenberg.org/files/27761/27761-0.txt'
PS> (new-object Net.WebClient).downloadString() | out-Speech
```

Pro dnes vše

Příště bych mohl představit ještě alespoň jednu sadu příkazů. Těžko se mi do dnešního dílu vybíralo, nabízelo se hodně možností. Alespoň jeden článek se tedy pravděpodobně bude věnovat PSCX a dalším zajímavým příkazům.

Čím jiným se rozloučit před Vánoci než 'merry christmas and happy new year' | out-speech (česky mu to bohužel nejde).

Minule jsme se seznámili s modulem [PSCX](#) a dnes ukážu, v jakých situacích se může modul hodit. Mimochodem, modul PSCX neuniknul ani oku [Scotta Hanselmana](#).

Předpokládejme, že pracuji na nějakém projektu, je vcelku jedno, na jakém. Potřebuji vygenerovat [Lorem ipsum](#), abych naplnil (HTML) stránku přibližně věrohodným obsahem.

Get-LoremIpsum

Pro daný účel poslouží cmdlet [Get-LoremIpsum](#). Pokud nespecifikujeme délku, výstupem je právě jeden odstavec. Pokud bychom chtěli text přesně dané délky, můžeme specifikovat počet znaků, slov, nebo odstavců.

```
PS> Get-LoremIpsum -Character 3
Lor
PS> Get-LoremIpsum -Word 3
Lorem ipsum dolor
PS> Get-LoremIpsum -Paragraph 3
Lorem ipsum dolor sit amet, ...
Duis autem vel eum ...
...
```

V tuto chvíli sice máme text, ale ještě je potřeba jej někam zkopírovat. Copy & Paste v konzoli je dost nepohodlné. Daleko lepší je použít cmdlet [Set-Clipboard](#). Výsledkem tedy je

```
PS> Get-LoremIpsum -p 1 | Set-Clipboard
```

Invoke-BatchFile

Čas od času je třeba spustit VS command line a z ní pak některý z toolků jako např. fuslogvw. Z prosté command line tool spustit nejde, možná jen proto, že cesta k němu není jen v PATH proměnné prostředí. Na import proměnných prostředí zavoláme cmdlet [Invoke-BatchFile](#) s cestou k bat souboru.

Cmdlet jednoduše spustí příslušný bat soubor předaný jako parametr a všechny proměnné prostředí existující po spuštění souboru naimportuje do PowerShell session. Příklad:

```
Invoke-BatchFile "${env:VS100COMNTOOLS}..\..\VC\vcvarsall.bat"
```

S touto myšlenkou přišel [Lee Holmes](#) a na jeho původní skript je možné se podívat na [poshcode.org](#). Všimněte si triku s `&& set` a přesměrováním do souboru, odkud se pak proměnné přečtou.

ConvertFrom-Base64, ConvertTo-Base64

Hlavně při práci na webu můžeme potřebovat pracovat s base64. Pak nám pomůžou cmdlety [ConvertFrom-Base64](#), [ConvertTo-Base64](#). Bohužel ale jako vstup neberou prostý *string*, ale je nutné konvertovat vstup na pole bytů.

```
PS> 'abc' | ConvertTo-Base64 # vyhodí chybu, bohužel.. je nutné konvertovat na pole bytů
```

```
PS> [byte[]][char[]]'proměnné prostředí.' | ConvertTo-Base64
```

Pokud ale použijeme české znaky, dostaneme chybu i tak:

```
PS> [byte[]][char[]]'proměnné prostředí.' | ConvertTo-Base64
```

```
Cannot convert value "ě" to type "System.Byte". Error: "Value was either too large or too small for an unsigned byte."
```

Musíme si pomoci jinak:

```
[text.encoding]::utf8.GetBytes('proměnné prostředí.') | ConvertTo-Base64
```

Cmdlety byly zamýšleny především pro kódování a dekódování souborů. V helpu můžete najít např. toto použití:

```
PS> $b64 = ConvertTo-Base64 Foo.dll -NoLineBreak
```

Resolve-WindowsError, Resolve-HResult

Řekněme, že už máme nějakou funkční aplikaci a ona najednou začne padat a jediná stopa je číslo chyby. První věc, kterou můžeme udělat před tím, než otevřeme prohlížeč, je spuštění jednoho z cmdletů (podle typu chyby).

Kódy můžeme do commandletů posílat přes pipeline, což nám umožní zpracovat více kódu najednou:

```
PS> 1..5 | Resolve-WindowsError
```

Nesprávná funkce

Systém nemůže nalézt uvedený soubor

Systém nemůže nalézt uvedenou cestu

Systém nemůže otevřít soubor

Přístup byl odepřen

Většina skriptů moc zábavná není, co tak si tedy skripty zpestřit nějakou náhodou hláškou? Náhodné hodnoty se generují pomocí **Get-Random**:

```
PS> 1..5 | % { Get-Random -min 0 -max 1000 } | Resolve-WindowsError
```

Neplatné atributy objektu zadané do funkce NtCreatePort nebo neplatné atributy portu zadané do funkce NtConnectPort

Unknown error (0x200)

Zařazovací vyrovnávací paměť uživatel/jádro přetekla

{Závažná chyba systému}

Bitová kopie systému %s není správně podepsána.

Soubor byl nahrazen podepsaným souborem.

```
System byl ukončen
Unknown error (0x3a1)
PS> 1..5 | % { Get-Random -min 0 -max 1000 } | Resolve-WindowsError
Unknown error (0x189)
Unknown error (0xf8)
Unknown error (0x210)
Disk je plný
Zadaná vyrovnávací paměť obsahuje samé nuly
```

Test-Xml, Format-Xml

Při mergování změn do *.proj souboru se někdy zadaří a výsledkem je nevalidí XML. Před commitem, nebo podobnou operací, kdy se chceme přesvědčit, že XML je validní, může posloužit **Test-Xml**. Jako parametr lze předat i odkaz na XSD, nebo DTD.

```
PS> gci D:\MicroTools\ *.fsproj -recurse | Test-Xml
True
True
True...
```

Pro zobrazení XML může posloužit již v minulém díle zmiňovaný **Format-Xml**.

```
PS> gci D:\MicroTools\ *.fsproj -rec | Format-Xml
```

echoargs

Dostali jsme se až k buildu. V rámci tohoto procesu můžeme volat z nějakého PowerShell skriptu nativní aplikace (např. msbuild). Toto je bohužel oblast, ve které často dochází k chybám. V některých případech je totiž velmi těžké správně doplnit apostrofy a escape sekvence. Důkazem budiž [několik](#) z [mnoha dotazů na StackOverflow](#).

Jako příklad si vezměme poslední dotaz:

```
PS> echoargs -verb:sync -source:dbfullsql="Data Source=mysource;Integrated
Security=false;User ID=sa;Pwd=sapass!;Database=mydb;" -dest:dbfullsql="Data Sourc
e=.\\mydestsource;Integrated Security=false;UserID=sa;Pwd=sapass!;Database=mydb;",
computername=10.10.10.10,username=administrator,password=adminpass
```

```
Arg 0 is <-verb:sync>
Arg 1 is <-source:dbfullsql=Data>
Arg 2 is <Source=mysource;Integrated>
Arg 3 is <Security=false;User>
Arg 4 is <ID=sa;Pwd=sapass!;Database=mydb;>
Arg 5 is <-dest:dbfullsql=Data>
```

```
Arg 6 is <Source=.\mydestsource;Integrated>
Arg 7 is <Security=false;User>
Arg 8 is <ID=sa;Pwd=sapass!;Database=mydb; computername=10.10.10.10 username=adm..
.

Command line:
"C:\Users\stej\Documents\WindowsPowerShell\Modules\pscX\Apps\EchoArgs.exe" -verb:
s
ync "-source:"dbfullsql=Data Source=mysource;Integrated Security=false;User ID=sa;
P
wd=sapass!;Database=mydb;" " "-dest:"dbfullsql=Data Source=.\mydestsource;Integrate
d
Security=false;User ID=sa;Pwd=sapass!;Database=mydb;" computername=10.10.10.10 use
r
name=administrator password=adminpass"
```

Na samém konci výpisu vidíme command line tak, jak ji vidí systém. Tento výpis je implementován v nejnovější verzi [2.1 Beta1](#). Dřívější verze obsahovaly pouze výpis argumentů.

Get-FileVersionInfo

Z práce nás vyruší email o spadlém buildu. Prý neprošly testy, protože se nám do binárek připletla assembly se špatnou verzí. V logu vidíme jasně, že nemohl načíst assembly *abc.dll*. V této chvíli potřebujeme rychle zjistit, jaké verze assembly vlastně v adresářích máme a kde se nachází ta špatná. Najdeme tedy všechny soubory daného jména a pošleme je do [Get-FileVersionInfo](#).

```
PS> gci D:\MicroTools\ *common*dll -rec | Get-FileVersionInfo
```

ProductVersion	FileVersion	FileName
0.5.0.18216	0.5.0.18216	D:\MicroTools\bin\MicroCommon.dll
0.5.0.18216	0.5.0.18216	D:\MicroTools\MicroCommon\obj\Debug\MicroCommon.dll
0.5.0.18216	0.5.0.18216	D:\MicroTools\pub\MicroCommon.dll
0.4.0.32450	0.4.0.32450	D:\MicroTools\old\MicroCommon.dll

Pokud víme, že některá verze na disku nemá co dělat, můžeme ji pak jednoduše smazat takto:

```
gci D:\MicroTools\ *common*dll -rec |
  Get-FileVersionInfo |
  ? { $_.ProductVersion -match '0\.4' } |
  Remove-Item -path {$_ .FileName}
```

Zde stojí za povšimnutí poslední řádek s [Remove-Item](#); jako argument se totiž předává ne hodnota, ale *skriptblock*, který se vyhodnotí pro každý objekt z pipeline a naváže se pak na parametr [-path](#). Ve *skriptblocku* se můžeme odkazovat na proměnnou [\\$_](#), která obsahuje aktuální prvek v pipeline. Jak je

tedy patrné, kód z objektu vráceného cmdletem `Get-FileVersionInfo` vrátí obsah property `FileName` a ten se naváže na parametr `-path`. Jiný (ale ne ekvivalentní) zápis by mohl být tento:

```
gci D:\MicroTools\ *common*.dll -rec |  
  Get-FileVersionInfo |  
  ? { $_.ProductVersion -match '0\.' } |  
  select -expand filename |  
  Remove-Item
```

První možnost je přehlednější a kratší. Naposledy o této možnosti psal [Jim Christopher](#).

Help

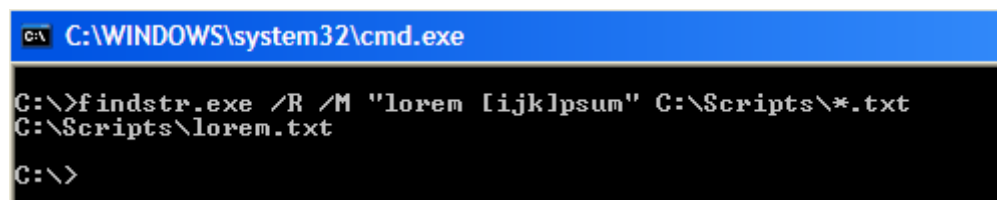
A v poslední řadě musím zmínit i příjemnější help. Pro porovnání si můžete zkusit `help gc -full` s importovaným PSCX a bez. PSCX přidává svou vlastní funkci `help`, která využívá `less.exe`. Můžeme se tak pohybovat po helpu šipkou nahoru/dolu a ukončit režim helpu klávesou `q`. Po opuštění zmizí obsah helpu a ukáže se původní obsah konzole. Příjemné, že?

Dnes jsme si ukázali, co nabízí modul `PSCX`. Nezapomeňte ovšem, že předvedené příkazy tvoří jen malou část z toho, co modul nabízí. Podívejte se na všechny příkazy, jistě najdete další zajímavé: `Get-Command -Module pscx`.

SELECT-STRING

Při našem [posledním setkání](#), jsme si ukázali základy práce s regulárními výrazy. Věnovali jsme se operátoru `-match` a dnes tyto znalosti využijeme při práci s cmdletem `Select-String`.

Určitě jste někdy potřebovali prohledávat textové soubory na výskyt klíčového slova. Ve Windows lze použít příkaz `findstr`. Řekněme, že hledáte známý text *lorem ipsum* v souborech v určitém adresáři.



```
C:\> C:\WINDOWS\system32\cmd.exe  
C:\> findstr.exe /R /M "lorem [ijkl]psum" C:\Scripts\*.txt  
C:\Scripts\lorem.txt  
C:\>
```

Jak je vidět i `findstr` používá regulární výraz (zde vyjádřeno pomocí parametru `R`). Zkusíme si předchozí příklad přepsat do PowerShellu.

```
PS C:\> Select-String 'lorem [ijk]psum' C:\Scripts\*.txt -List |  
Select Path
```

Path

C:\Scripts\lorem.txt

Vidíte, že v případě použití PowerShellu se `findstr` jeví o něco méně ukecaná varianta. Nesmíme ovšem zapomínat, že v PowerShellu pracujeme s objekty a výsledky můžeme dále zpracovávat v rouře. Více si povíme v další části článku.

Základní použití Select-String

Zkusíme získat aktuální IP adresu počítače z výpisu příkazu *ipconfig*.

```
PS C:\> ipconfig
```

```
Windows IP Configuration
```

```
Ethernet adapter Wireless Network Connection 3:
```

```
Connection-specific DNS Suffix . :  
IP Address. . . . . : 192.168.20.203  
Subnet Mask . . . . . : 255.255.255.0  
Default Gateway . . . . . : 192.168.20.200
```

Potřebujeme získat řádku, která obsahuje text „IP Address“, Vzhledem k tomu, že *ipconfig* vrací pouze text:

```
PS C:\> ipconfig | gm
```

```
TypeName: System.String
```

```
... zkráceno
```

použijeme nejjednodušší variantu cmdletu **Select-String**.

```
PS C:\> ipconfig | Select-String 'ip address'
```

```
IP Address. . . . . : 192.168.20.203
```

Vidíme, že **Select-String** je v základní podobě *case insensitive* a můžeme tedy hledaný výraz psát malými písmeny. Pokud bychom hledali *case sensitive*, musíme použít parametr **CaseSensitive**.

Jelikož jsme si říkali, že vše je objekt, pojďme se podívat na výsledek předchozí řádky z pohledu výsledného objektu:

```
PS C:\> ipconfig | Select-String 'ip address' | fl * -Force
```

```
IgnoreCase : True  
LineNumber : 15  
Line : IP Address. . . . . : 192.168.20.203  
Filename : InputSteam  
Path : InputSteam  
Pattern : ip address  
Context :  
Matches : {IP Address}
```

Hned na první řádce vidíme, že **IgnoreCase** je opravdu nastaveno na **True**. Dále jsou pro nás zajímavé vlastnosti **Line**, **Pattern** a **Matches**. Vzhledem k tomu, že vyhledáváme řádky obsahující určitý text, pro další zpracování nás bude nejvíc zajímat vlastnost **Line**.

```
PS C:\> $1 = (ipconfig | Select-String 'ip address').Line.Trim()  
PS C:\> $1  
IP Address. . . . . : 192.168.20.203
```

Zde jsme využili jednu z vynikajících vlastností PowerShellu. Pokud uzavřeme příkaz do závorek:

```
(ipconfig | Select-String 'ip address')
```

Chová se tento jako objekt a k jeho vlastnostem můžeme přistupovat pomocí tečkové notace. Zápis:

```
$1 = (ipconfig | Select-String 'ip address').Line
```

Je tedy ekvivalentní zápisu

```
$12 = ipconfig | Select-String 'ip address' | Select -Expand Line
```

Jelikož jsou na začátku řádku mezery, které se nám pro další zpracování nehodí, použili jsme metodu **Trim()**, abychom tyto mezery odstranili.

Pro získání IP adresy použijeme operátor **-match**:

```
PS C:\> $1 -match '(?<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})$'  
True  
PS C:\> $matches.ip  
192.168.20.203
```

Vzor pro vyhledání IP adresy je jeden z nejjednodušších, který existuje. Není potřeba vymýšlet již vymyšlené. Vynikajícím zdrojem pro regulární výrazy je např. adresa <http://regexlib.com>.

Je vidět, že jednoduchým postupem jsme získali aktuální IP adresu počítače. Metod, pomocí kterých získáme lokální IP adresu, je veliké množství. Některé využívají WMI třídu *Win32_NetworkAdapterConfiguration*, jiné používají .NET třídu *System.Net.Dns*. Je čistě na vás, jakou metodu byste v tomto případě preferovali. Předem upozorňuji na to, že naznačený postup bude fungovat, pokud máte IP adresu přiřazenou pouze pro jeden adaptér.

Vzhledem k tomu, že jsme pro vyhledávání adresy nepoužili žádný regulární výraz, stačilo by nám použít parametr **SimpleMatch**.

```
PS C:\> ipconfig | Select-String 'ip address' -SimpleMatch  
  
IP Address. . . . . : 192.168.20.203
```

Pokud bychom například chtěli získat všechny řádky, které obsahují IP adresu, **SimpleMatch** bychom nemohli použít. **Select-String** funguje standardně tak, že regulární výrazy používá.

```
PS C:\> ipconfig | Select-String
'(?<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})'
```

```
IP Address. . . . . : 192.168.20.203
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.20.200
```

Dále bych výstup mohli zpracovat libovolným způsobem na principu první ukázky.

Prohledávání logů

Dále budeme pracovat s logem WindowsUpdate.log. Co třeba zobrazit si verzi klienta Windows Update. Můžeme se podívat, jestli se verze klienta v únoru (2012-02) změnila

```
Get-Content c:\WINDOWS\WindowsUpdate.log | Select-String '^2012-
02.*WU client version'
2012-02-12 20:51:52:796 1044 c50 Agent * WU client version
7.4.7600.226
2012-02-13 21:25:55:343 1040 6a4 Agent * WU client version
7.4.7600.226
2012-02-17 22:34:53:046 1040 cdc Agent * WU client version
7.4.7600.226
2012-02-18 11:21:26:843 1040 14c Agent * WU client version
7.4.7600.226
2012-02-19 11:46:17:250 1232 7b4 Agent * WU client version
7.4.7600.226
2012-02-19 15:48:13:156 1244 780 Agent * WU client version
7.4.7600.226
2012-02-19 21:11:23:406 1048 ab0 Agent * WU client version
7.4.7600.226
...zkráceno
```

Select-String umožňuje zobrazit i kontext, ve kterém se hledaný řetězec vyskytuje. Struktura log souboru v okolí hledaného textu je následující:

```
2012-02-24 08:12:07:593 1068 6d0 Misc ===== Logging
initialized (build: 7.4.7600.226, tz: +0100) =====
2012-02-24 08:12:07:640 1068 6d0 Misc = Process:
C:\WINDOWS\System32\svchost.exe
2012-02-24 08:12:07:640 1068 6d0 Misc = Module:
C:\WINDOWS\system32\wuaueng.dll
2012-02-24 08:12:07:578 1068 6d0 Service *****
2012-02-24 08:12:07:640 1068 6d0 Service ** START ** Service:
Service startup
2012-02-24 08:12:07:640 1068 6d0 Service *****
2012-02-24 08:12:08:015 1068 6d0 Agent * WU client version
```

```

7.4.7600.226
2012-02-24 08:12:08:015 1068 6d0 Agent * Base directory:
C:\WINDOWS\SoftwareDistribution
2012-02-24 08:12:08:031 1068 6d0 Agent * Access type: No proxy
2012-02-24 08:12:08:171 1068 6d0 Agent * Network state: Connected
2012-02-24 08:12:55:906 1068 6d0 Agent ***** Agent:
Initializing Windows Update Agent *****
2012-02-24 08:12:55:921 1068 6d0 Agent ***** Agent:
Initializing global settings cache *****
2012-02-24 08:12:55:921 1068 6d0 Agent * WSUS server: <NULL>
2012-02-24 08:12:55:921 1068 6d0 Agent * WSUS status server: <NULL>
2012-02-24 08:12:55:921 1068 6d0 Agent * Target group: (Unassigned
Computers)
2012-02-24 08:12:55:921 1068 6d0 Agent * Windows Update access
disabled: No
2012-02-24 08:12:56:031 1068 6d0 Agent * Found 1 persisted download
calls to restore

```

Pokud chceme zobrazit stejný počet řádek před a za hledaným textem, použijeme parametr **Context** s jedním argumentem:

```

PS C:\> Get-Content c:\WINDOWS\WindowsUpdate.log | Select-String
'^2012-02.*WU client version' -Context 2

```

```

2012-02-12 20:51:52:531 1044 c50 Service ** START ** Service:
Service startup
2012-02-12 20:51:52:578 1044 c50 Service *****
> 2012-02-12 20:51:52:796 1044 c50 Agent * WU client version
7.4.7600.226
2012-02-12 20:51:53:140 1044 c50 Agent * Base directory:
C:\WINDOWS\SoftwareDistribution
2012-02-12 20:51:53:140 1044 c50 Agent * Access type: No proxy
2012-02-13 21:25:55:328 1040 6a4 Service ** START ** Service:
Service startup
2012-02-13 21:25:55:328 1040 6a4 Service *****
> 2012-02-13 21:25:55:343 1040 6a4 Agent * WU client version
7.4.7600.226
2012-02-13 21:25:55:343 1040 6a4 Agent * Base directory:
C:\WINDOWS\SoftwareDistribution
2012-02-13 21:25:55:343 1040 6a4 Agent * Access type: No proxy
...zkráceno

```

Znak > ukazuje řádku s hledaným textem a vidíme dvě řádky před a za ním. V případě, že chceme vidět specifický počet řádek před/za textem, použijeme dva argumenty:

```

PS C:\> Get-Content c:\WINDOWS\WindowsUpdate.log | Select-String
'^2012-02.*WU client version' -Context 0,3

```

```

> 2012-02-12 20:51:52:796 1044 c50 Agent * WU client version
7.4.7600.226
2012-02-12 20:51:53:140 1044 c50 Agent * Base directory:
C:\WINDOWS\SoftwareDistribution
2012-02-12 20:51:53:140 1044 c50 Agent * Access type: No proxy
2012-02-12 20:51:53:406 1044 c50 Agent * Network state: Connected
> 2012-02-13 21:25:55:343 1040 6a4 Agent * WU client version
7.4.7600.226
2012-02-13 21:25:55:343 1040 6a4 Agent * Base directory:
C:\WINDOWS\SoftwareDistribution
2012-02-13 21:25:55:343 1040 6a4 Agent * Access type: No proxy
2012-02-13 21:25:55:343 1040 6a4 Agent * Network state: Connected

```

Znak > nám opět ukazuje hledanou řádku a vidíme, že kontext se zobrazuje pouze za ní (tři následující řádky).

Prohledávání více souborů

Velmi častým případem bude prohledávání více souborů. Můžeme si například zobrazit všechny výskyty textu „failed“ v log souborech pro jednotlivé záplaty.

```
PS C:\> ls c:\Temp\kb*.log | Select-String 'failed'
```

```

Temp\KB2412687.log:5:2.250: Failed To Enable SE_SHUTDOWN_PRIVILEGE
Temp\KB2412687.log:7:2.250: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
Temp\KB2412687.log:46:3.266: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
Temp\KB2412687.log:85:8.375: LoadFileQueues:
UpdSpGetSourceFileLocation for halmacpi.dll failed: 0xe0000102
Temp\KB2485663.log:5:2.891: Failed To Enable SE_SHUTDOWN_PRIVILEGE
Temp\KB2485663.log:7:2.891: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
Temp\KB2485663.log:78:3.375: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
Temp\KB2497640-IE8.log:5:2.640: Failed To Enable
SE_SHUTDOWN_PRIVILEGE
Temp\KB2497640-IE8.log:7:2.734: In Function GetReleaseSet, line
1240, RegQueryValueEx failed with error 0x2
Temp\KB2497640-IE8.log:49:3.594: In Function GetReleaseSet, line
1240, RegQueryValueEx failed with error 0x2
Temp\KB2503658.log:6:2.046: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
Temp\KB2503658.log:7:2.484: DoInstallation: CleanPFR failed: 0x2
...zkráceno

```

Vidíte, že **Select-String** vypíše jednotlivé řádky obsahující hledaný text (včetně čísla řádky) a jméno souboru, ve kterém se text vyskytuje. Pokud bychom chtěli vidět pouze první výskyt hledaného textu v souborech, použili bychom parametr **List**.

```
PS C:\> ls c:\Temp\*.log | Select-String 'failed' -List
```

```
Temp\KB2412687.log:5:2.250: Failed To Enable SE_SHUTDOWN_PRIVILEGE
Temp\KB2485663.log:5:2.891: Failed To Enable SE_SHUTDOWN_PRIVILEGE
Temp\KB2497640-IE8.log:5:2.640: Failed To Enable
SE_SHUTDOWN_PRIVILEGE
Temp\KB2503658.log:6:2.046: In Function GetReleaseSet, line 1240,
RegQueryValueEx failed with error 0x2
...zkráceno
```

Select-String je vynikající cmdlet pro práci s texty. V některých případech je lepší využít operátor `-match`, ale pro hledání informací ve velkém množství souborů (např. logy – viz příklad) je **Select-String** ideální variantou.

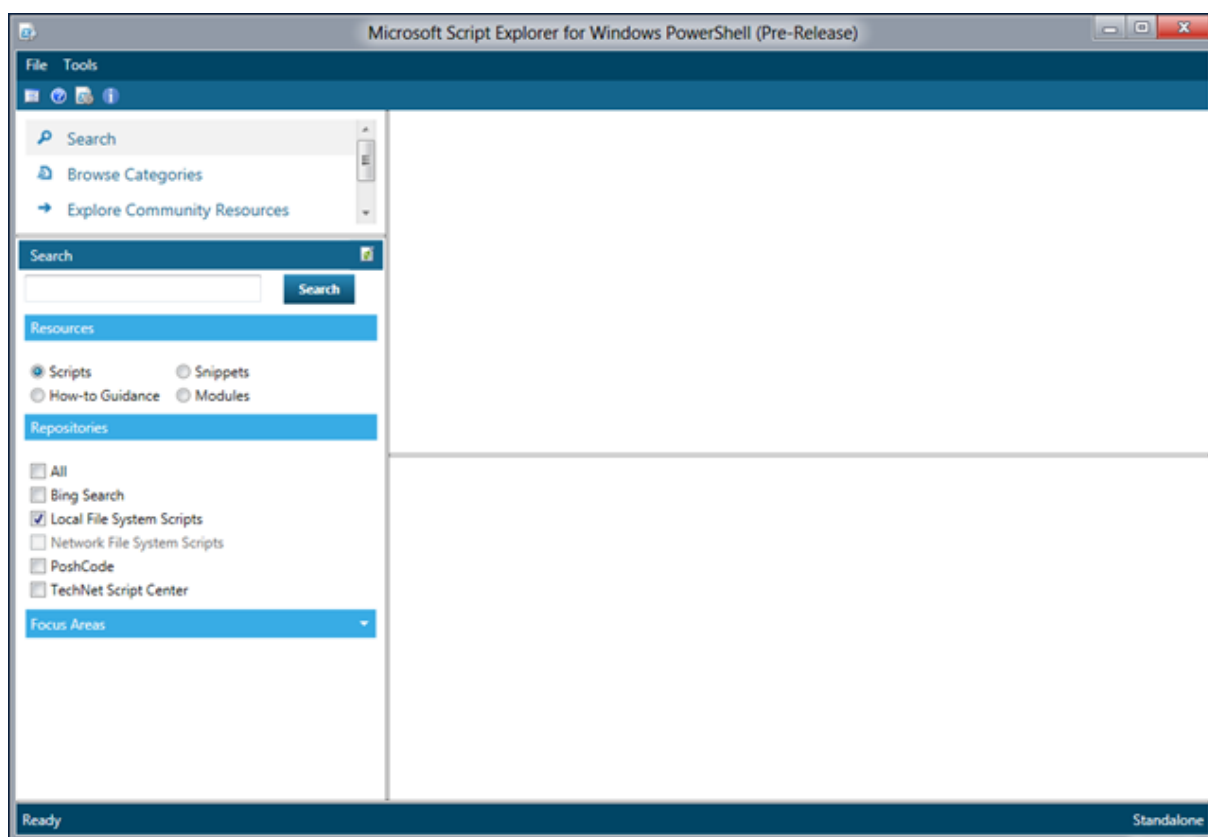
MICROSOFT SCRIPT EXPLORER FOR WINDOWS POWERSHELL

Při mých prezentacích na konferenci [TechDays](#) se (právem) těšil největší přízni lidí Microsoft Script Explorer for Windows PowerShell (dále jen Script Explorer). Tento malý program umožňuje vyhledávat skripty (ale i moduly) v předem daných lokacích. V současné době například na vašem lokálním disku, na síťovém úložišti nebo na internetu. Momentálně je k dispozici prohledávání např. TechNet Script Center nebo PoShCode.org – dle mého asi nejlepší zdroje skriptů pro PowerShell.



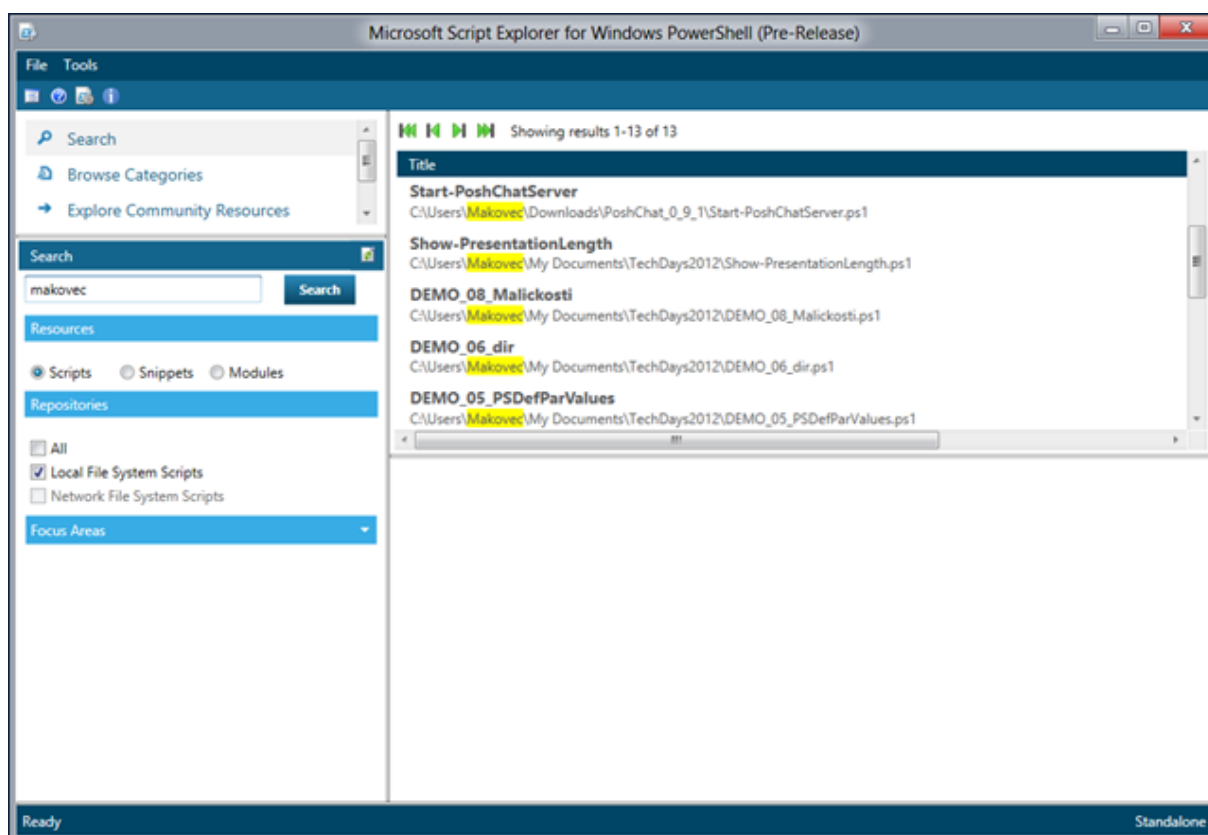
Od doby TechDays se objevila beta Script Exploreru a té bych se v dnešním článku rád věnoval. Script Explorer je dostupný na stránkách [Download centra](#) a na stránkách microsoft.com má i vlastní stránky s velice dobře zpracovanou nápovědou. Tyto stránky najdete na adrese <http://scriptexplorer.microsoft.com>.

Po spuštění se objeví základní okno:

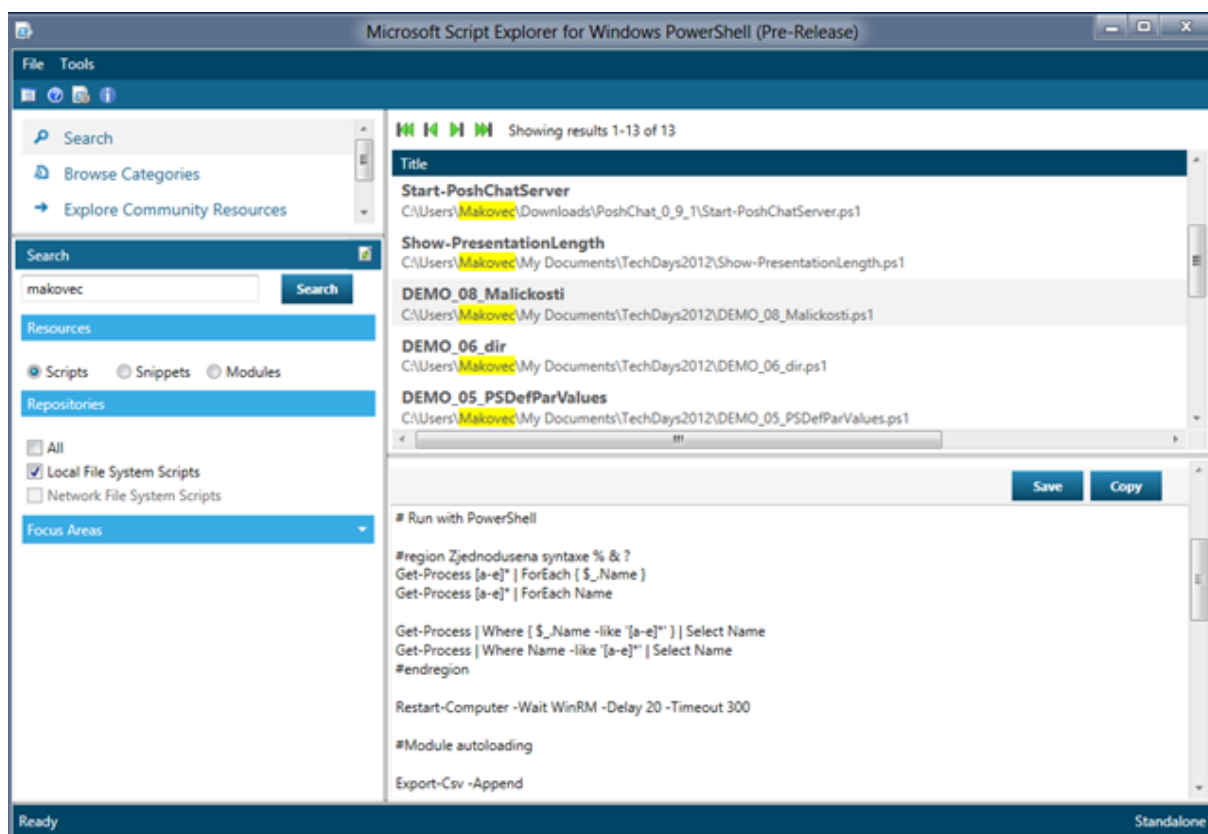


Toto okno je rozděleno do několika částí. V levém sloupci můžete ovlivňovat to, jaké kategorie budete hledat a také to, v jaké zdroje budete prohledávat. Vidíte, že v sekci **Resources** jsou dostupné čtyři možnosti a kategorie **Repositories** obsahuje například i prohledávání výsledků vyhledávače Bing.

Pokud necháme zaškrtnuté pouze prohledávání *Local File System Scripts*, výsledkem budou dle očekávání pouze skripty na našem počítači:



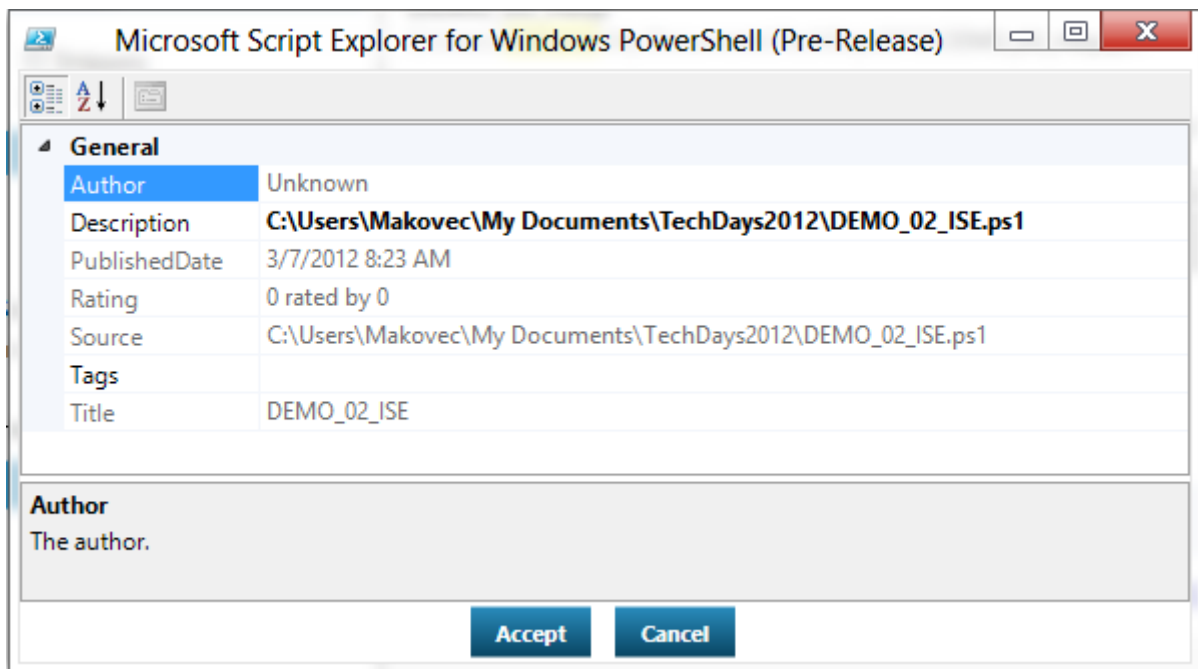
V panelu výsledků vidíte část, které odpovídá hledanému textu. Script Explorer samozřejmě prohledává i kód ve skriptech, takže pokud by se hledaný text „makovec“ objevil uvnitř skriptu (například v komentáři), celý skript by byl ve výsledcích také vidět. Pokud si některý ze skriptů vybereme, jeho obsah se objeví v pravém dolním panelu:



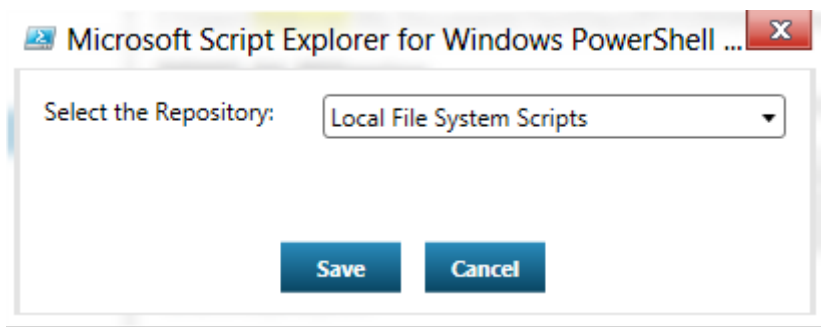
Pomocí tlačítka *Copy* uložíme skript do schránky a můžeme jej použít v konzoli nebo ISE (pro práci v ISE si ale za chvíli ukážeme lepší možnost).

Tlačítkem *Save* můžeme skript uložit. Při stahování z internetu se Script Explorer pro jistotu zeptá, jestli jsme si jisti a poté nám ve dvou oknech nabídne uložení.

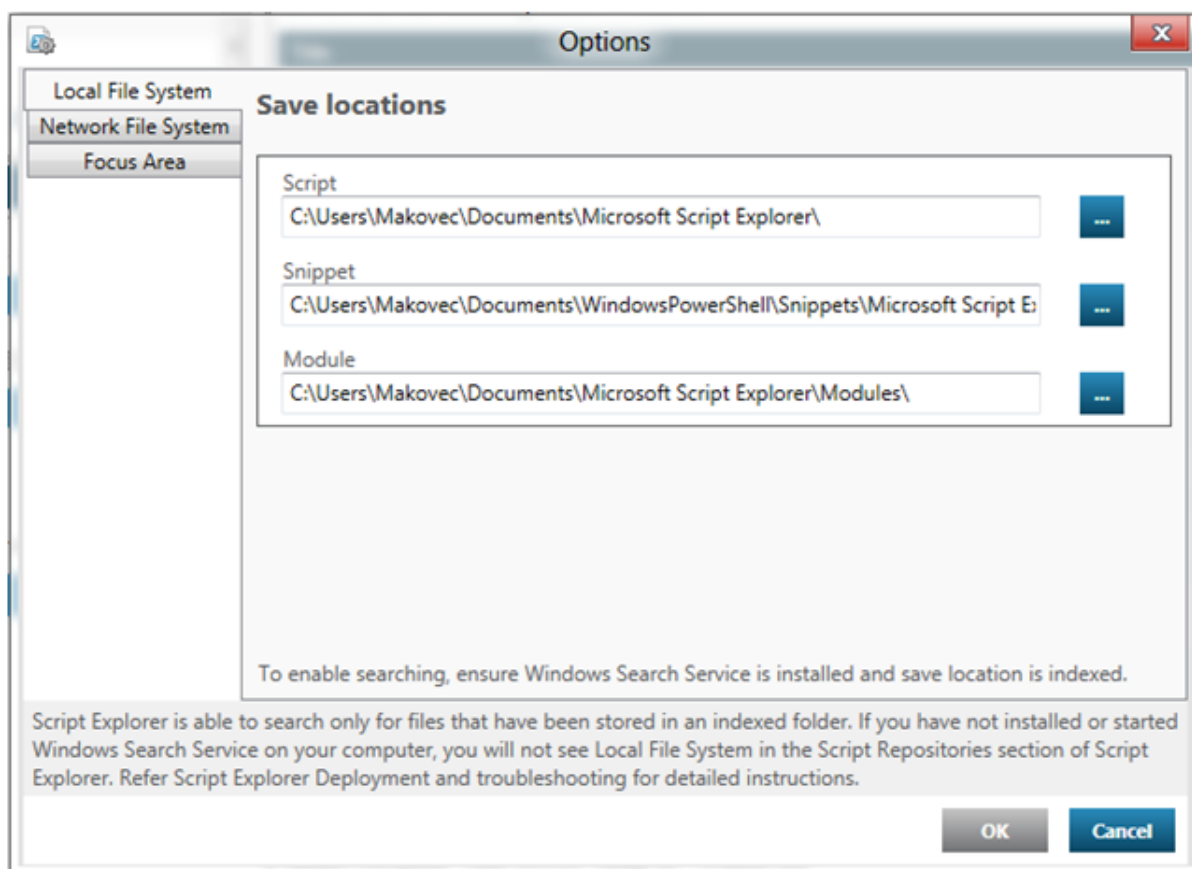
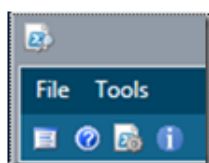
V prvním okně se zobrazí podrobnější informace o skriptu:



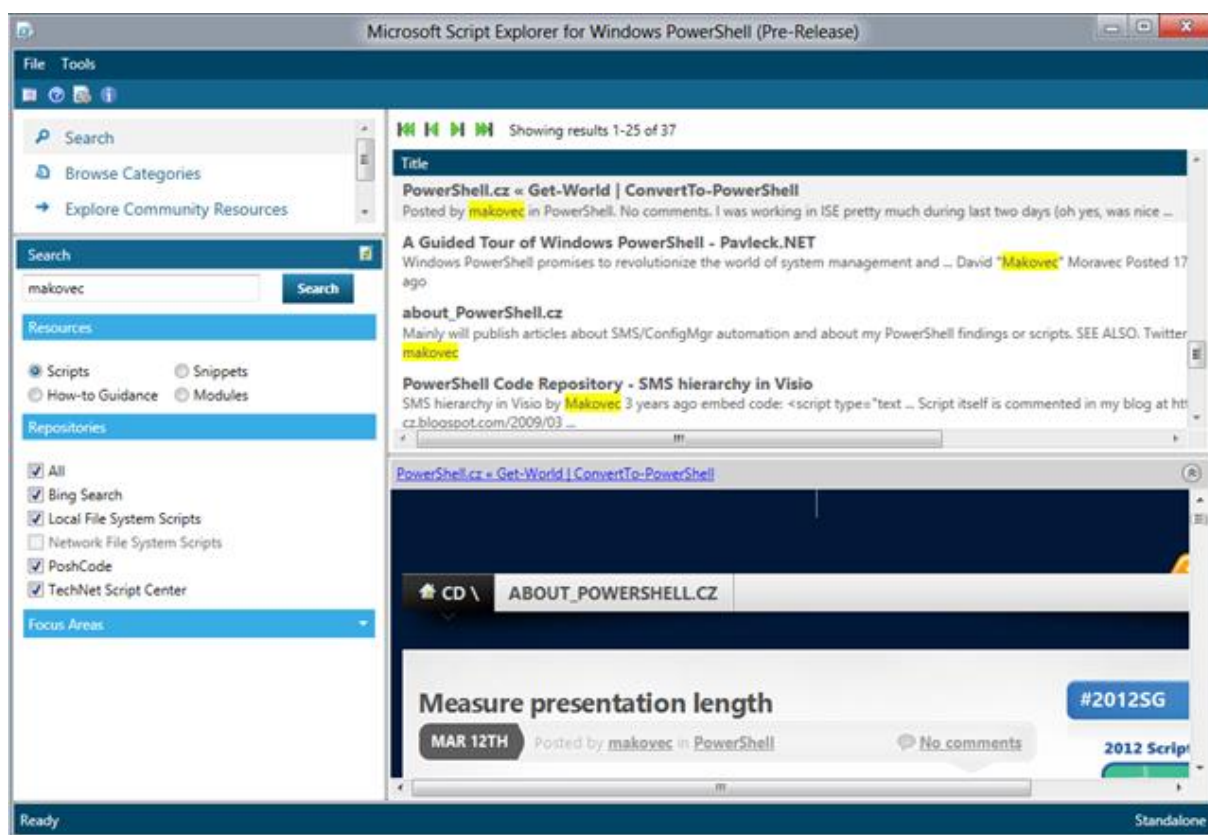
A ve druhém již můžeme zvolit, kam budeme skript ukládat:



V nastavení Script Exploreru můžeme zvolit, jaké adresáře budou prohledávané při volbě *Local File System Scripts*. Nastavení je dostupné přes třetí ikonu v horní liště:

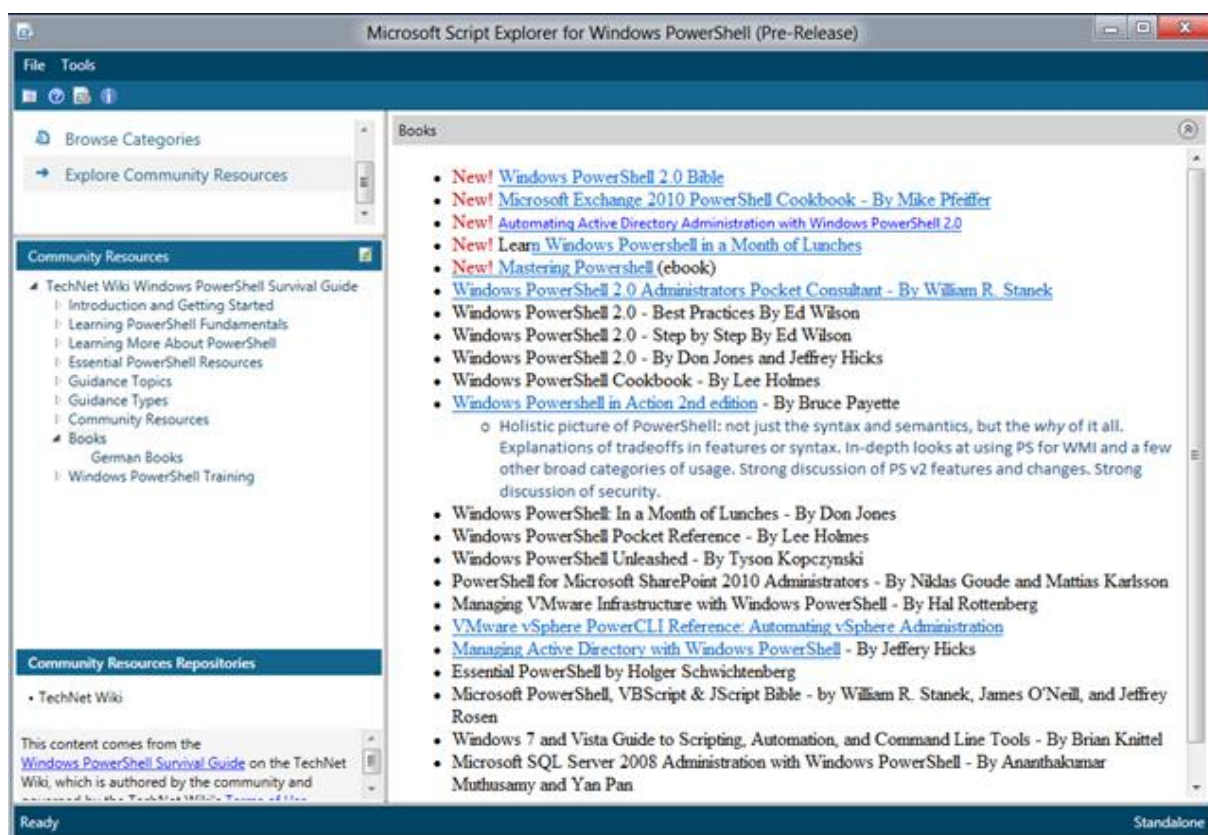


Vidíte, že můžete nastavovat jak lokální cestu ke skriptům, tak i síťové úložiště. Pojdme si ještě ukázat, jak funguje prohledávání zdrojů na internetu. Postup je naprosto shodný s vyhledáváním lokálním. Pouze musíte v části Repositories zvolit prohledávaný zdroj:

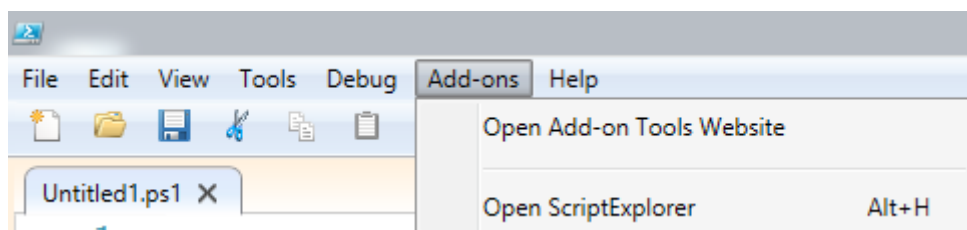


Vidíte, že výsledek je očekávaný. Zde jsem využil zajímavé možnosti vyhledávání ve výsledcích Bingů a mám zobrazen jeden článek ze svého blogu. Script Explorer mi tímto umožňuje stát se jediným nástrojem pro hledání jakéhokoli skriptu či modulu, který budu potřebovat hledat.

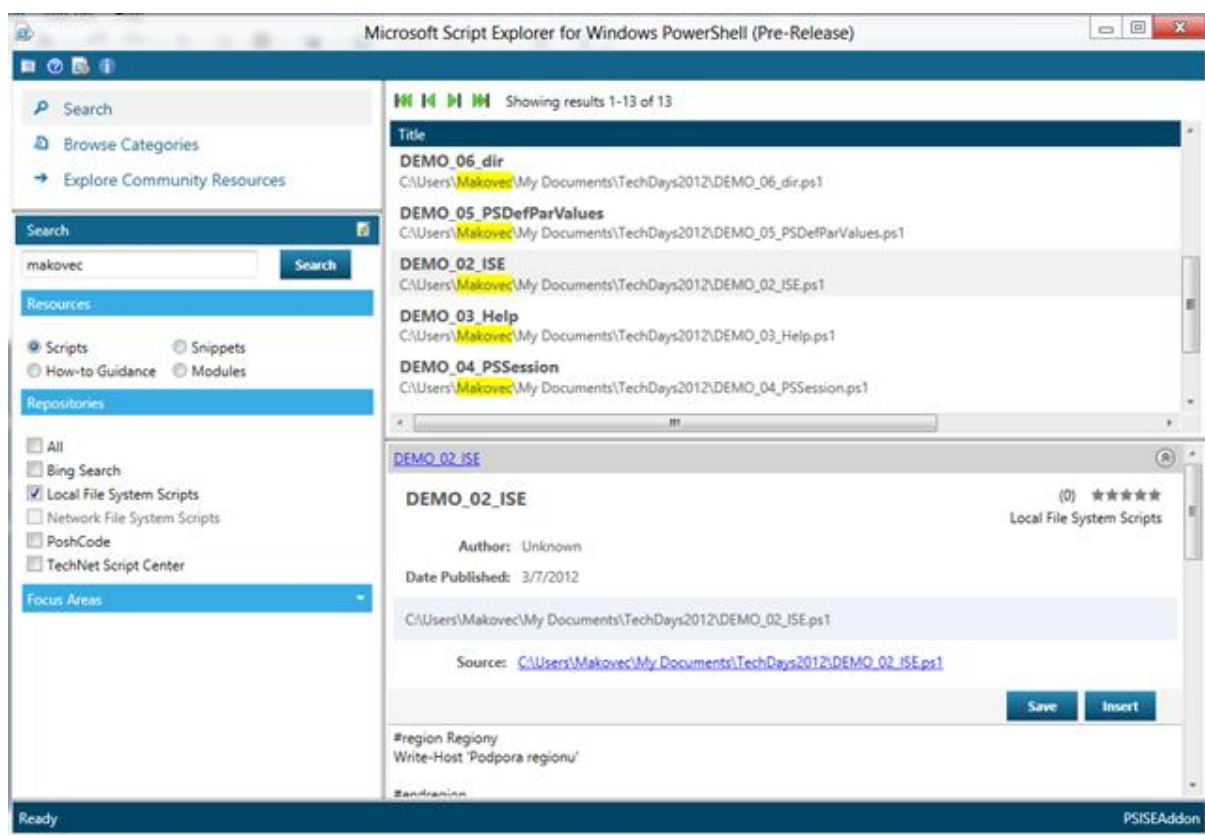
Zajímavou možností Script Exploreru je i část **Explore Community Resources**. Tato sekce odkazuje na množství článků o PowerShellu a vy tedy získáváte další možnost, jak z jednoho nástroje přistoupit např. k nápovědě PowerShellu:



A poslední možností, o které bych se chtěl dnes zmínit je integrace Script Exploreru do ISE. Pokud si zaregistrujete Script Explorer jako Add-On a přiřadíte jeho spuštění nějakou klávesovou zkratku,



můžete jej volat přímo z ISE. Výhodou je v tomto případě hlavně nové tlačítko *Insert*, kterým pak můžete daný skript vložit přímo do ISE:



Script Explorer je v současné době dostupný pro klienty Windows Vista SP2, Windows 7 SP1 a Windows 8 Consumer Preview. Serverové systémy můžete dohledat v linku na začátku tohoto článku.

Vzhledem k tomu, že jsem měl možnost být u toho, když Script Explorer vznikal, mám k němu trochu citový vztah:). Myslím si, že se jedná o skvělý doplněk, který by neměl chybět ve výbavě nikomu, kdo pracuje s PowerShellem častěji a ve svých skriptech se začíná ztrácet. Zároveň je to i skvělý nástroj pro sdílení skriptů v týmu. Rozhodně jej doporučuji k vyzkoušení.

OPERÁTOR FORMÁTOVÁNÍ

V nedávno skončených [Scripting Games](#) se často vyskytoval ve výsledných skriptem jeden „nešvar“ – výstup byl často podáván ve formě textu pomocí cmdletu Write-Host. Obecně proti Write-Host nic nemám, ale v pravidlech Scripting Games (a v diskusích) bylo často zmiňováno, že výstupem by měly být objekty. My dnes ale půjdeme opačnou cestou a podíváme se, jak vytvořit textový výstup pomocí operátoru formátování (-f).

Pozor: PowerShell je nad objekty postaven a práce s objekty je u něm přirozená. Ovšem občas se nám ale hodí na výstupu textová informace nebo prostě jenom chceme generovat velké množství textu.

Operátor formátování nám říká, **jak** chceme formátovat a **co** chceme formátovat. Vezměme si následující příklad:

```
PS C:\> $proc = Get-Process powershell
PS C:\> $proc
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
254	7	57780	57672	162	2.55	3856	powershell

```
PS C:\> "Process {0} ma Id {1}" -f $proc.Name, $proc.Id
Process powershell ma Id 3856
```

```
PS C:\> Write-Host "Process $proc.Name ma Id $proc.Id"
Process System.Diagnostics.Process (powershell).Name ma Id
System.Diagnostics.Process (powershell).Id
```

```
PS C:\> Write-Host "Process $($proc.Name) ma Id $($proc.Id)"
Process powershell ma Id 3856
```

Do proměnné `$proc` jsme si uložili aktuální proces PowerShellu a na následujícím řádku jsme vypsalí jeho jméno a Id pomocí operátoru formátování (k zápisu se hned vrátím). Vidíte, že na následujících dvou řádkách jsem zkusil vypsat tu samou informaci pomocí cmdletu `Write-Host`. V prvním jsem získal z mého pohledu naprosto neužitečnou informaci a teprve ve druhém správná data. Musel jsem totiž použít tzv. *subexpression*, což je příkaz PowerShellu uzavřený v kulatých závorkách uvozených znakem dolaru: **`$(<sem_patří_nějaký_příkaz>)`**

Pokud PowerShell vidí *subexpression* v textovém řetězci, provede nejdříve příkaz v závorce a teprve poté poskládá celý řetězec. V našem příkladu se tedy nejprve převedly `$proc.Name` a `$proc.Id` na správné texty a pak se celý řetězec složil do očekávané podoby.

A nyní zpátky k operátoru formátování. Na levé straně máme uvedeno, jak chceme formátovat:

```
"Process {0} ma Id {1}"
```

Čísla ve složených závorkách mi určují místa, kam přijdou proměnné z pravé strany operátoru formátování. Číslování je stejně jako u polí, uváděno o nuly, čili první proměnná je zobrazena symbolem `{0}`. Na pravé straně pak máme ony zmiňované proměnné:

```
$proc.Name, $proc.Id
```

které se nám ve výsledném řetězci přeloží na požadované hodnoty. Celý zápis

```
"Process {0} ma Id {1}" -f $proc.Name, $proc.Id
```

bychom tedy mohli přechíst jako: *Vypiš text „Process“, dále vlož jméno procesu, text „ma Id“ a následně Id zmiňovaného procesu*. Pokud bychom například chtěli vypsat stejnou informaci pro více procesů, můžeme to provést následujícím způsobem:

```
PS C:\> Get-Process | Select -First 3 | ForEach { "Process {0} ma Id
{1}" -f $_.Name, $_.Id }
Process AESTFltr ma Id 3104
Process ApMsgFwd ma Id 3904
Process ApntEx ma Id 924
```

Pro první tři procesy na mém počítači jsem vypsal požadovanou informaci. Pořadí vypisovaných informací můžeme samozřejmě libovolně měnit, můj oblíbený příklad je tento:

```
PS C:\> "{4}{4}{1}{5}{0}{3}{0}{5}{2}" -f 'e','ka','l','p','po','t'
```

Dáte dohromady výsledek? Ze zápisu je vidět, že jsem některé proměnné (zde na pravé straně zastoupené daným textem) použil vícekrát a rozhodně jsem nedodržel dané pořadí na levé straně. Nicméně v případě takového zápisu už bych trochu pochyboval o čitelnosti. J Mimochodem, výsledný text je *popokatepetl*.

Formátování ve formátování

V příkladu s výpisem více procesů jste si všimli, že výsledek není úplně oku lahodící. Někdy by se nám hodilo, aby se jednotlivé položky výstupu zarovnávaly pod sebe. Naštěstí operátor formátování disponuje možností dalšího formátování (proto ten divný nadpis). Zmiňovaný příklad můžeme tedy přepsat například takto:

```
PS C:\> Get-Process | Select -First 3 | ForEach { "Process '{0,8}'  
ma Id: {1,4}" -f $_.Name, $_.Id }  
Process 'AESTFltr' ma Id: 3104  
Process 'ApMsgFwd' ma Id: 3904  
Process ' ApntEx' ma Id: 924
```

Pro lepší představu jsem do výstupu přidal jednoduché uvozovky. Zápis {0,8} říká, že nahrazovaný text bude dlouhý osm znaků a v případě, že bude kratší, bude doplněn zleva mezerami. To samé pravidlo se uplatní i v případě Id daného procesu ({1,4}), kde šířka textu bude čtyři znaky. Pokud bych použil větší číslo, výsledek bude následující:

```
PS C:\> "Process '{0,15}' ma Id: {1,8}" -f $proc.Name, $proc.Id  
Process ' powershell' ma Id: 3856
```

Možná je ještě o trochu lepší způsob zarovnat text (jméno procesu) ve vytvořeném bloku na levou stranu. V tom případě použijeme šířku textu se zápornou hodnotou:

```
PS C:\> Get-Process | Select -First 3 | ForEach { "Process {0,-10}  
ma Id: {1,4}" -f $_.Name, $_.Id }  
Process AESTFltr ma Id: 3104  
Process ApMsgFwd ma Id: 3904  
Process ApntEx ma Id: 924
```

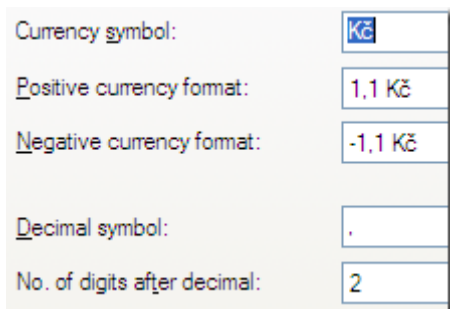
Další možností úpravy je formátování pomocí *formátovacího řetězce*. Podíváme se na nejjednodušší formátování, například měny a poté se vrátíme k formátování čísel.

```
PS C:\> $name=50  
PS C:\> $nakup=63.5
```

```
PS C:\> "Kdyz budeme mit {0} a utratime {1}, zustane nam {2}." -f  
$name, $nakup, ($name-$nakup)  
Kdyz budeme mit 50 a utratime 63.5, zustane nam -13.5.
```

```
PS C:\> "Kdyz budeme mit {0:C} a utratime {1:C}, zustane nam {2:C}."
-f $mame, $nakup, ($mame-$nakup)
Kdyz budeme mit 50,00 Kč a utratime 63,50 Kč, zustane nam -13,50 Kč.
```

Do proměnných si uložíme částku, kterou máme na účtu a kterou pak utratíme. V prvním případě výsledný text nijak neformátujeme a dostáváme větu jak z automatického překladače. Ve druhém případě doplníme za znak dvojtečky formátovací řetězec pro zápis měny (Currency). Vidíte, že výsledek se bez jakéhokoli dalšího přispění změnil na vcelku srozumitelnou podobu mých výdajů a příjmů. Formátování hodnot je dáno nastavením prostředí Windows, takže v mém případě takto:



V závěrečném příkladu si ukážeme možnost formátování čísel. Příklad bude trochu složitější, nejdříve si ukážeme výsledek:

ProcessName	cpu	ws
TOTALCMD	465.0	11.89
System	260.3	1.26
WINWORD	213.8	77.27
procexp	184.5	26.39
svchost	125.1	99.32
powershell_ise	101.0	164.04
explorer	98.9	30.96
svchost	75.8	29.45
aenrjpro	69.9	8.82
chrome	68.0	23.44

Výsledkem je TOP10 procesů podle využití CPU.

Začneme tedy získáním procesů.

```
PS C:\> $procesy = Get-Process | Sort-Object CPU -Descending |
Select-Object -First 10
PS C:\> $procesy
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
196	7	6152	12160	62	519.73	4508	TOTALCMD
1104	0	0	1288	3	264.09	4	System
1150	21	65180	79224	274	222.66	3512	WINWORD

513	13	20812	27036	112	189.61	3556	procexp
706	50	71340	101704	542	125.13	2548	svchost
432	18	173456	168744	463	108.33	2952	
powershell_ise							
518	18	22552	31700	131	100.33	3372	explorer
2470	146	17576	29688	154	76.73	856	svchost
199	8	20864	24012	122	70.36	3388	chrome
276	9	7532	9036	74	69.94	1180	aenrjpro

Nyní si dáme dohromady hlavičku, kterou použijeme ve výpisu.

```
PS C:\> $header = "{0,-20}{1,6}{2,7}" -f 'ProcessName','cpu','ws'
PS C:\> $header = "{0}`n{1}" -f $header, ('-'*$header.Length)
PS C:\> $header
ProcessName          cpu          ws
-----
```

Již dopředu jsme si rozmysleli, jména a šířku jednotlivých sloupců. První řádka určuje jména sloupců a druhá řádka vytváří „čáru“ pro oddělení hlavičky od vlastních hodnot.

Poznámka: Všimněte si jedné zajímavosti – pokud násobíte dvě proměnné a první z nich je typu string (řetězec), bude výsledkem opakování tohoto řetězce, tedy

```
PS C:\> $a='xxxxxy'
PS C:\> $a*3
xxxxxyxxxxxyxxxxxy
```

```
PS C:\> $b='123'
PS C:\> $b*3
123123123
PS C:\> 3*$b
369
```

V posledním příkladu jsem jako první uvedl číslo a proto se celý výraz provádí jako násobení dvou čísel. Proto jsem v mé definované hlavičce mohl využít zápisu ('-'*\$header.Length) pro opakované vypsání znaku -. Tím bychom měli definovanou hlavičku našeho výpisu. Nyní budeme formátovat vlastní data, podívejme se na následující zápis:

```
PS C:\> $c = 1.12345
PS C:\> "{0:###.0}`n{0:###.00}" -f $c
1.1
1.12
```

Počet desetinných míst dané proměnné je zobrazen v závislosti na počtu desetinných míst ve formátu. Toho můžeme využít při zarovnávání ve výsledné tabulce. Připravíme si proto jednotlivé části a vyzkoušíme je na naší proměnné z našeho dnešního prvního příkladu.

```
PS C:\> $proc
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	
ProcessName							
-----	-----	-----	-----	-----	-----	---	-----
-							
470	7	56152	57788	163	4.95	3856	powershell

```
PS C:\> $p = "{0,-20}" -f $proc.ProcessName
PS C:\> $cpu = "{0,6:###.0}" -f $proc.CPU
PS C:\> $ws = "{0,7:###.00}" -f ($proc.WS/1MB)
PS C:\> "{0}{1}{2}" -f $p, $cpu, $ws
powershell 4.8 56.43
```

První formátování je nám již známe – určujeme šířku sloupce pro jméno procesu. Ve druhém a třetím kombinujeme šířku sloupce se zobrazením na určitý počet desetinných míst. Pro ověření můžeme samozřejmě vyzkoušet toto:

```
PS C:\> $cpu
4.8
PS C:\> $ws
56.43
PS C:\> $proc
powershell
PS C:\> $proc.Length
20
```

A vidíte, že jednotlivé položky jsou opravdu „předformátované“ podle naší potřeby. Nyní vše spojíme dohromady.

```
$procesy = Get-Process | Sort-Object CPU -Descending | Select-Object
-First 10
$header = "{0,-20}{1,6}{2,7}" -f 'ProcessName','cpu','ws'
$header = "{0}\n{1}" -f $header, ('-' * $header.Length)
$header
foreach ($p in $procesy) {
    $proc = "{0,-20}" -f $p.ProcessName
    $cpu = "{0,6:###.0}" -f $p.CPU
    $ws = "{0,7:###.00}" -f ($p.WS/1MB)
    "{0}{1}{2}" -f $proc, $cpu, $ws
}
```

A výsledkem bude náš očekávaný výstup:

ProcessName	cpu	ws
-----	-----	-----
TOTALCMD	465.0	11.89
System	260.3	1.26
WINWORD	213.8	77.27
procexp	184.5	26.39
svchost	125.1	99.32

powershell_ise	101.0	164.04
explorer	98.9	30.96
svchost	75.8	29.45
aenrjpro	69.9	8.82
chrome	68.0	23.44

Formátovací operátor se dá využít například pro generování textových souborů v určitém formátu. Připravíte si jednotlivé „šablony“ jako v předchozím případě a pak výsledný text pouze přesměrujete na potřebný výstup.

TVORBA VLASTNÍCH OBJEKTŮ

V [minulém díle](#) jsme vytvářeli textový výstup pomocí operátoru formátování. Dnes se podíváme na opačný pól – vytvoříme si vlastní objekt z daného textového výstupu.

Již několikrát jsem zmiňoval, že PowerShell je postaven nad objekty a s objekty dokáže velice efektivně pracovat. Problém občas nastává, pokud vytváříte vlastní skript nebo funkci a na výstupu chcete mít vlastní objekty. V tomto případě se hodí cmdlet **New-Object**. Tento cmdlet má jeden povinný parametr, **TypeName**, tento parametr určuje typ objektu – ve většině našich případů zde zadejte [PSObject](#). Tím vytvoříte objekt typu System.Management.Automation.PSCustomObject – bohužel vytvoření objektu bez vlastností nebo metod nedává moc smysl:

```
PS C:\> New-Object -TypeName PSObject
```

```
PS C:\> $a = New-Object -TypeName PSObject
PS C:\> $a
```

```
PS C:\> $a.GetType().FullName
System.Management.Automation.PSCustomObject
PS C:\> $a | Get-Member
```

```
TypeName: System.Management.Automation.PSCustomObject
Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
```

K tomuto objektu můžeme vlastnosti přidávat později za běhu skriptu, například takto:

```
[20]: $a = $a | Add-Member -MemberType NoteProperty -Name Vlastnost -Value
'Nejaky text' -PassThru
[21]: $a
```

```
Vlastnost
-----
Nejaky text
```

```
[22]: $a | Get-Member
```

```
      TypeName: System.Management.Automation.PSCustomObject
Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
```

```
ToString      Method      string ToString()
Vlastnost     NoteProperty System.String Vlastnost=Nejaky text
```

Pomocí cmdletu **Add-Member** jsme k původnímu objektu přidali vlastnost **Vlastnost** a přiřadili jsme jí určitou hodnotu. I když je tento způsob vytváření objektů možný, existuje naštěstí lepší volba. Tou je použití parametru **Parameter** při vytváření objektu pomocí **New-Object**. Pojďme si vytvořit nový objekt:

```
PS C:\> New-Object -TypeName PSObject -Property @{ a=1; b=2; c=3 }
```

```
a      b      c
-      -      -
1      2      3
```

```
PS C:\> $b = New-Object -TypeName PSObject -Property @{ a=1; b=2; c=3 }
PS C:\> $b | Get-Member
```

```
TypeName: System.Management.Automation.PSCustomObject
```

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
a	NoteProperty	System.Int32 a=1
b	NoteProperty	System.Int32 b=2
c	NoteProperty	System.Int32 c=3

Parameter očekává hash tabulku, jejíž dvojice klíč-hodnota se stanou vlastností a její hodnotou v novém objektu. Hash tabulku si samozřejmě můžeme vytvořit předem a pak ji uvést jako hodnotu parametru.

```
PS C:\> $prop = @{
>> a=1
>> b=2
>> c=3
>> }
>>
PS C:\> $prop
```

```
Name      Value
-----
a          1
b          2
c          3
```

```
PS C:\> $prop.GetType().FullName
System.Collections.Hashtable
PS C:\> New-Object -TypeName PSObject -Property $prop
```

```
a      b      c
-      -      -
1      2      3
```

Vidíte, že tvorba nového objektu proběhla ve dvou krocích. Nejdříve jsme v hash tabulce určili, jaké vlastnosti bude nový objekt obsahovat a ve druhém kroku jsme tyto vlastnosti použili při vytváření objektu typu PSObject. Tím jsme si oddělili „starosti“ s přemýšlením o vlastnostech objektu. Zkusme jednoduchý příklad.

```
function Get-TNObject
{
```

```

param(
    $ComputerName = $env:COMPUTERNAME
)

$os = Get-WmiObject -ComputerName $ComputerName -Class
Win32_OperatingSystem

$prop = @{
    ComputerName = $ComputerName
    OSName       = $os.Caption
    OSVersion    = $os.Version
    ScanTime     = Get-Date
}

New-Object -TypeName PSObject -Property $prop
}

```

Funkce Get-TNObject očekává na vstupu jméno počítače, který testujeme. Uvnitř funkce zavoláme cmdlet **Get-WmiObject** a zjistíme informace o operačním systému. Do hash tabulky poté vložíme informace o jménu počítače, jeho operačním systému a verzi a také datum, kdy jsme tento sken spustili. Na posledním řádku poté vytváříme objekt. Tento objekt je dále pouštěn do roury a výsledky funkce tam můžeme zpracovat jiným cmdletem. Použití je následující:

```
PS C:\> Get-TNObject
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	05-Jun-12 11:29:14	Microsoft Windows XP Profe...	MAKOVEC-40

```
PS C:\> Get-TNObject localhost
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	05-Jun-12 11:29:18	Microsoft Windows XP Profe...	localhost

```
PS C:\> Get-TNObject localhost | Format-Table -AutoSize
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	05-Jun-12 11:29:32	Microsoft Windows XP Professional	localhost

Pokud bychom chtěli zpracovávat více počítačů, můžeme použít například cmdlet **ForEach-Object**.

```
PS C:\> 'localhost',$env:COMPUTERNAME,'makovec-40','192.168.100.100' |% { Get-TNObject $_; Sleep 2 }
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	05-Jun-12 11:34:59	Microsoft Windows XP Profe...	localhost
5.1.2600	05-Jun-12 11:35:01	Microsoft Windows XP Profe...	MAKOVEC-40
5.1.2600	05-Jun-12 11:35:03	Microsoft Windows XP Profe...	makovec-40
5.1.2600	05-Jun-12 11:35:05	Microsoft Windows XP Profe...	192.168.100.100

Pro kontrolu jsem přidal čekání dvě vteřiny, aby bylo vidět, že ScanTime se nám opravdu mění v závislosti na volání funkce. Jak jsem již zmiňoval – výstupem jsou objekty a můžeme s nimi dále libovolně pracovat.

```
PS C:\> 'localhost',$env:COMPUTERNAME, 'makovec-40','192.168.100.100' |% { Get-TNObject $_ } | Sort ComputerName
```

OSVersion	ScanTime	OSName	ComputerName
-----------	----------	--------	--------------

```
5.1.2600 05-Jun-12 11:38:28 Microsoft Windows XP Profe... 192.168.100.100
5.1.2600 05-Jun-12 11:38:28 Microsoft Windows XP Profe... makovec-40
5.1.2600 05-Jun-12 11:38:28 Microsoft Windows XP Profe... MAKOVEC-40
5.1.2600 05-Jun-12 11:38:28 Microsoft Windows XP Profe... localhost
```

```
PS C:\> 'localhost',$env:COMPUTERNAME, 'makovec-40','192.168.100.100' |% {
Get-TNObject $_ } | Sort ComputerName | ft -AutoSize
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	05-Jun-12 11:38:43	Microsoft Windows XP Professional	192.168.100.100
5.1.2600	05-Jun-12 11:38:43	Microsoft Windows XP Professional	makovec-40
5.1.2600	05-Jun-12 11:38:43	Microsoft Windows XP Professional	MAKOVEC-40
5.1.2600	05-Jun-12 11:38:43	Microsoft Windows XP Professional	localhost

Tato technika vytváření objektů za běhu funkce nebo skriptu je velice užitečná. Vzhledem k tomu, že v PowerShellu pracujeme s objekty velice často, byla by škoda, pokud by vaše funkce/skripty pouze posílaly textový výstup pomocí **Write-Host**. Až budete příště zpracovávat nějaká data, vzpomeňte si na **New-Object** a zkuste jej využít.

Typ objektu

Při našem [posledním předprázdninovém setkání](#) jsme vytvářeli vlastní objekty. Dnes posuneme naši snahu o dokonalost o trochu dále. Pouze pro jistotu zopakujeme použitý kód:

```
function Get-TNObject
{
    param(
        $ComputerName = $env:COMPUTERNAME
    )

    $os = Get-WmiObject -ComputerName $ComputerName -
ClassWin32_OperatingSystem

    $prop = @{
        ComputerName = $ComputerName
        OSName       = $os.Caption
        OSVersion    = $os.Version
        ScanTime     = Get-Date
    }

    New-Object -TypeName PSObject -Property $prop
}
```

Výstupem dané funkce je například:

```
PS C:\> Get-TNObject | Format-Tablet -AutoSize
```

OSVersion	ScanTime	OSName	ComputerName
5.1.2600	03-Sep-12 23:23:48	Microsoft Windows XP Professional	MAKOVEC-40

Pokud si uložíme výstupní objekt do proměnné – se kterou chceme dále pracovat - dostaneme následující výsledky:

```
PS C:\> $tn = Get-TNObject
```

```
PS C:\Scripts > $tn.GetType()
```

IsPublic	IsSerial	Name	BaseType
True	False	PSCustomObject	System.Object

```
PS C:\> $tn.GetType().FullName
System.Management.Automation.PSCustomObject
```

```
PS C:\Scripts > $tn | gm
```

```
TypeName: System.Management.Automation.PSCustomObject
```

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ComputerName	NoteProperty	System.String ComputerName=MAKOVEC-40
OSName	NoteProperty	System.String OSName=Microsoft Windows XP Professional
OSVersion	NoteProperty	System.String OSVersion=5.1.2600
ScanTime	NoteProperty	System.DateTime ScanTime=0š-Sep-12 23:25:46

Vidíme, že objekt typu **PSCustomObject**. Vzhledem k tomu, že si dnes budeme trochu hrát s typem objektu, hodilo by se nám zjistit, jak jsme se vlastně dostali k „výslednému“ typu objektu.

Pozn.: Pokud jsou mezi vámi vývojáři, promiňte prosím v rámci zjednodušení některé prohršky proti běžným konvencím. Dopouštím se jich vědomě.

```
PS C:\> $tn.PSTypeNames
System.Management.Automation.PSCustomObject
System.Object
```

Pokud čteme výstup odspodu, vidíme, že náš TN objekt je zároveň typu **Systém.Object** (základ všech objektů v .NET) a již uvedeného **PSCustomObject**. Tato vlastnost se nám může hodit v případě vytváření vlastních rozšíření pomocí formátovacích souborů (o těchto souborech si povíme více v některém z příštích pokračování). Nejprve si ale musíme připravit naše objekty. Pro názornost si ještě ukážeme, jakého typu je například objekt z WMI.

```
PS C:\> $os = Get-WmiObject -Class Win32_OperatingSystem
PS C:\> $os.PSTypeNames
System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem
System.Management.ManagementObject
System.Management.ManagementBaseObject
System.ComponentModel.Component
System.MarshalByRefObject
System.Object
```

Vidíte, že opět začínáme u **Systém.Object** a pokračujeme dalšími typy až k výsledném Win32_OperatingSystem.

Přemýšleli jste někdy nad tím, proč se při zadání

```
PS C:\> Get-WmiObject Win32_OperatingSystem

SystemDirectory : C:\WINDOWS\system32
Organization    : PowerShell.cz
BuildNumber     : 2600
RegisteredUser  : PowerShell.cz
SerialNumber    : 12345-678-12345-678910
Version        : 5.1.2600
```

zobrazí pouze pár vlastností, když jich je ve skutečnosti mnohem více?

```
PS C:\> Get-WmiObject Win32_OperatingSystem | Get-Member -MemberType Property  
| Measure-Object
```

Count : 71

Pro zobrazení všech hodnot můžete použít

```
PS C:\> Get-WmiObject Win32_OperatingSystem | Format-List -Property *
```

Nebudeme zde z důvodu místa uvádět výsledek – sami si ho můžete jednoduše zobrazit. Zobrazení vlastností je kontrolované právě takzvaným formátovacím souborem. Pokud si zobrazíme část tohoto souboru – právě pro WMI objekty – uvidíme toto:

```
<Type>  
  <Name>System.Management.ManagementObject#root\cimv2\win32_OperatingSystem</Name>  
  <Members>  
    <PropertySet>  
      <Name>PSStatus</Name>  
      <ReferencedProperties>  
        <Name>Status</Name>  
        <Name>Name</Name>  
      </ReferencedProperties>  
    </PropertySet>  
    <PropertySet>  
      <Name>FREE</Name>  
      <ReferencedProperties>  
        <Name>FreePhysicalMemory</Name>  
        <Name>FreeSpaceInPagingFiles</Name>  
        <Name>FreeVirtualMemory</Name>  
        <Name>Name</Name>  
      </ReferencedProperties>  
    </PropertySet>  
    <MemberSet>  
      <Name>PSStandardMembers</Name>  
      <Members>  
        <PropertySet>  
          <Name>DefaultDisplayPropertySet</Name>  
          <ReferencedProperties>  
            <Name>SystemDirectory</Name>  
            <Name>Organization</Name>  
            <Name>BuildNumber</Name>  
            <Name>RegisteredUser</Name>  
            <Name>SerialNumber</Name>  
            <Name>Version</Name>  
          </ReferencedProperties>  
        </PropertySet>  
      </Members>  
    </MemberSet>  
  </Members>  
</Type>
```

Pokud se podíváte na část **PSStandardMembers** a porovnáte ji s výstupem cmdletu `Get-WmiObject`, uvidíte jistou shodu. Právě tato část určuje, jaké vlastnosti se zobrazí při standardním výstupu.

Pozor! Pokud neodoláte touze a soubor najdete, v žádném případě jej neměňte. Soubor je digitálně podepsán a jeho změnou si narušíte jednu ze základních funkcí PowerShellu. Existují lepší metody, jak standardní výpis změnit.

Pojďme tedy na začátek dnešního článku a zkusme si změnit (přidat) nový typ k našemu TN objektu. Osobně mám rád cestu, kdy nejdříve zrušíme veškeré informace o typu a poté přidáme pouze náš vlastní.

```
PS C:\> $tn.PSTypeNames  
System.Management.Automation.PSCustomObject  
System.Object
```

```
PS C:\> $tn.PSTypeNames.Clear()
```

```
PS C:\> $tn.PSTypeNames
```

```
PS C:\> $tn.PSTypeNames.Add('Makovec.TNObjekt')
```

```
PS C:\> $tn.PSTypeNames  
Makovec.TNObjekt
```

```
PS C:\> $tn | Get-Member
```

```
TypeName: Makovec.TNObjekt
```

...

Nyní máme připravený objekt pro další práci. Abychom nemuseli upravovat výsledné objekty, je lepší měnit typ ihned po vytvoření objektu. Proto si trochu upravíme původní funkci.

```
function Get-TNObject  
{  
    param(  
        $ComputerName = $env:COMPUTERNAME  
    )  
  
    $os = Get-WmiObject -ComputerName $ComputerName -Class  
Win32_OperatingSystem  
  
    $prop = @{  
        ComputerName = $ComputerName  
        OSName       = $os.Caption  
        OSVersion    = $os.Version  
        ScanTime     = Get-Date  
    }  
  
    $obj = New-Object -TypeName PSObject -Property $prop  
    $obj.PSTypeNames.Clear()  
    $obj.PSTypeNames.Add('Makovec.TNObjekt')  
    $obj  
}
```

Při zobrazování tohoto typu objektu můžeme použít formátovací soubor. Jak již bylo uvedeno – použití tohoto souboru si ukážeme v některém z příštích pokračování.

Příště se podíváme na PowerShell v3 – vzhledem k tomu, že uvedení Win8 se blíží, je myslím vhodná doba pro opuštění v2. Nicméně vše, co jsme si ukazovali ve všech předchozích dílech, platí i ve verzi 3.

POWERSHELL V3

Jak jsem slíbil [minule](#), podíváme se dnes na novou verzi PowerShellu – v3. Nejprve krátce něco k dostupnosti na různých verzích Windows.

PowerShell v3 je standardní součástí Windows 8 a Windows Serveru 2012 (všechny edice). Doinstalovat jej lze na starší operační systémy následovně:

- Windows 7 SP1
- Windows Server 2008 SP2
- Windows Server 2008 R2 SP1

Stejně jako v předchozí verzi, je i nyní PowerShell součástí balíčku nazvaného Windows Management Framework 3.0. Tento balík obsahuje ještě WMI a WinRM. Pro instalaci je potřeba mít nainstalován .NET Framework 4.0. Více se můžete dočíst přímo na stránce, kde se dá [WMF stáhnout](#).

A nyní již k některým novinkám. Je jasné, že v dnešním článku nelze pokrýt všechny, ale vybral jsem ty, které považuji za nejzajímavější z mého pohledu. Samozřejmě si o dalších povíme v následujících dílech TechNet Flashe.

\$PSDefaultParameterValues

Tuto nově zavedenou proměnnou mám mezi svými TOP 3. Vezměte si následující situaci. Používáte často cmdlet **Export-Csv** a nelíbí se vám, že je na začátku každého souboru uvedena informace o exportovaném typu, například:

```
PS C:\> Get-Process powershell | Select-Object Name, Company, Id | Export-Csv
C:\temp\proc.csv
PS C:\> Get-Content C:\temp\proc.csv
#TYPE Selected.System.Diagnostics.Process
"Name","Company","Id"
"powershell","Microsoft Corporation","7040"
```

Export-Csv obsahuje parametr **NoTypeInfoInformation**, takže můžete použít

```
PS C:\> Get-Process powershell | Select Name, Company, Id | Export-Csv
C:\temp\proc.csv -NoTypeInfoInformation
```

a ve výsledném souboru se typ neobjeví. Jenomže uvádění parametru může být po delším čase otravné. Abychom nemuseli tento parametr uvádět, můžeme využít proměnnou

\$PSDefaultParameterValues. Ta určuje vaši vlastní hodnotu parametru pro daný cmdlet. V našem případě můžeme určit, že parametr **NoTypeInfoInformation** bude u cmdletu **Export-Csv** nastaven na True a nebudeme ho tedy muset uvádět.

```
$PSDefaultParameterValues = @{ 'Export-Csv:NoTypeInfoInformation'=$True }
```

Na příkladu je vidět, že definujeme hash tabulku, kde uvádíme jméno cmdletu, parametr a jeho (novou) standardní hodnotu. Toto nastavení je platné po dobu běhu PowerShellu, takže je ideální umístit změny do vašeho profilu. Pokud tedy nyní zkusíme znovu předchozí příklad

```
PS C:\> $PSDefaultParameterValues = @{ 'Export-Csv:NoTypeInfoInformation'=$True }
PS C:\> Get-Process powershell | Select-Object Name, Company, Id | Export-Csv
C:\temp\proc.csv
PS C:\> Get-Content C:\temp\proc.csv
"Name","Company","Id"
"powershell","Microsoft Corporation","7040"
```

Vidíte, že i bez uvedení **NoTypeInfoInformation**, máme ve výsledném souboru informace bez uvedení typu. Já mám například ve svém profilovém souboru pro **Export-Csv** následující záznam:

```
$PSDefaultParameterValues = @{
    'Export-Csv:Delimiter'=';';
    'Export-Csv:NoTypeInfoInformation'=$true
}
```

tím určuji nejen, že nechci ukládat informace o typu, ale zároveň určuji, jaký znak použiji jako oddělovač. Jak víte, v českém prostředí je standardně používán středník. Předchozím zápisem mám zajištěno, že veškeré mé výstupy do CSV použiji jako oddělovač právě středník a Excel jiných lidí si s ním poradí bez problémů.

To byla jen krátká ukázka použití, pokud vás zajímá více – a já myslím, že by mělo – podívejte se na nápovědu dostupnou na <http://technet.microsoft.com/en-us/library/hh847819.aspx>

Update-Help

Jak je vidět i v banneru mistra Skriptíka, **Update-Help** může způsobovat lehkou bolest hlavy. Pojdme se na tuto novinku podívat podrobněji. Již od PowerShellu verze 1 byl problém s neaktuální nápovědou. Z různých historických důvodů nebylo možné po uvolnění produktu jakýmkoli způsobem

nápovědu aktualizovat. Ve verzi 2 jsme měli možnost použít parametr **Online** u cmdletu **Get-Help**. Nicméně soubory nainstalované lokálně na počítači jsme opět nemohli aktualizovat. Proto přišel PowerShell tým s možností aktualizovatelné nápovědy (v originále updatable help).

Na čistě nainstalovaných Windows (lépe řečeno – po instalaci PowerShellu) se po zavolání **Get-Help** zobrazí nápověda pouze v základním tvaru, generovaném z metadat cmdletu. Pro aktualizaci je potřeba (jako administrátor) zavolat **Update-Help**. Tento cmdlet následně provede několik akcí:

- Zjistí všechny instalované moduly (dívá se do cesty **PSModulePath**).
- Podle **HelpUri** zjistí cestu k online verzi nápovědy (v **PSMAML** formátu).
- Zjistí, jestli je nainstalována poslední verze nápovědy.
- Stáhne CAB soubor nápovědy (pokud je potřeba) a rozbalí jej.
- Ověří, zda jsou soubory správné.
- Nainstaluje soubory do správných adresářů.

Shodou okolností byla hostem jednoho z posledních dílů [PowerScripting Podcastu June Blender](http://powerscripting.wordpress.com/2012/10/02/episode-203-june-blender-from-microsoft-talks-about-getting-help/). Ta je zodpovědná za tvorbu nápovědy v PowerShellu již mnoho let. Pokud si chcete tento díl poslechnout, najdete jej na adrese <http://powerscripting.wordpress.com/2012/10/02/episode-203-june-blender-from-microsoft-talks-about-getting-help/>

June mimo jiné zmiňovala, že **Update-Help** patří (pro mne překvapivě) k nejsložitějším cmdletům v PowerShellu. Více si o možnostech aktualizace nápovědy můžete přečíst TechNet stránkách věnovaných jednotlivým cmdletům: **Update-Help**, **Save-Help**, **Get-Help** nebo v tematické nápovědě [about_Updatable_Help](#).

Pro mne osobně je ideální scénář, kdy o víkendu spustíte automaticky **Save-Help** a jednotlivé počítače mají nastavenou v PowerShell profilu kontrolu pomocí **Update-Help** například každé pondělí. Tím máte zajištěnou dostupnost aktuální verze nápovědy na vašich počítačích.

Nové možnosti vytváření vlastních objektů

V předchozích dílech jsme se věnovali tvorbě vlastních objektů. Ukažme si jednoduchý příklad.

```
PS C:\> New-Object -TypeName PSObject -Property @{
>> a='jedna'
>> b='dva'
>> c='tri'
>> d='ctyri'
>> } | Format-Table -AutoSize
>>
```

a	b	d	c
-	-	-	-
jedna	dva	ctyri	tri

toto chování by nás již nemělo překvapit. Pokud bychom chtěli poradit vlastností tak, jak jsme je v objektu definovali, bylo by potřeba použít cmdlet **Select-Object**

```
PS C:\> New-Object -TypeName PSObject -Property @{
>> a='jedna'
>> b='dva'
>> c='tri'
>> d='ctyri'
>> } | Select-Object a,b,c,d | Format-Table -AutoSize
>>
```

a	b	c	d
-	-	-	-
jedna	dva	tri	ctyri

Naštěstí existuje ve verzi 3 lepší možnost – použití třídy **PSCustomObject**:

```
PS C:\> [PSCustomObject] @{
>> a='jedna'
>> b='dva'
>> c='tri'
>> d='ctyri'
>> } | Format-Table -AutoSize
>>
```

a	b	c	d
jedna	dva	tri	ctyri

Pro mne osobně je toto velké ulehčení práce, protože u všech mnou vytvářených objektů chci mít zajištěné pořadí jednotlivých vlastností.

Stejná technika jde použít i v případě vytváření objektů, např. třídy **PSSessionOption**:

```
[System.Management.Automation.Remoting.PSSessionOption]@{IdleTimeout=43200000;
SkipCnCheck=$True}
```

A tento objekt můžeme dále využít v cmdletech využívajících tento typ objektu.

Dnes jsme si ukázali úplný začátek naší cesty po PowerShellu v3. Pokud by vás zajímala nějaká specifická část nového PowerShellu, nechte komentář pod článkem, příště se můžeme věnovat i vašemu tématu.

POWERSHELL V3 – DALŠÍ ZAJÍMAVÉ NOVINKY

Restart-Computer

Mnohokrát se vám stane, že potřebujete ve vašem skriptu nebo funkci restartovat vzdálený počítač a po restartu pokračovat další akcí. Typickým případem je například přidání počítače do domény a jeho následující konfigurace. V PowerShellu verze 2 se toto řešilo typicky ve smyčce, kde jste prováděli ping na vzdálený počítač a po nějaké době jste ve skriptu pokračovali dále. Nová verze naštěstí přináší nové parametry: **Wait** a **For**.

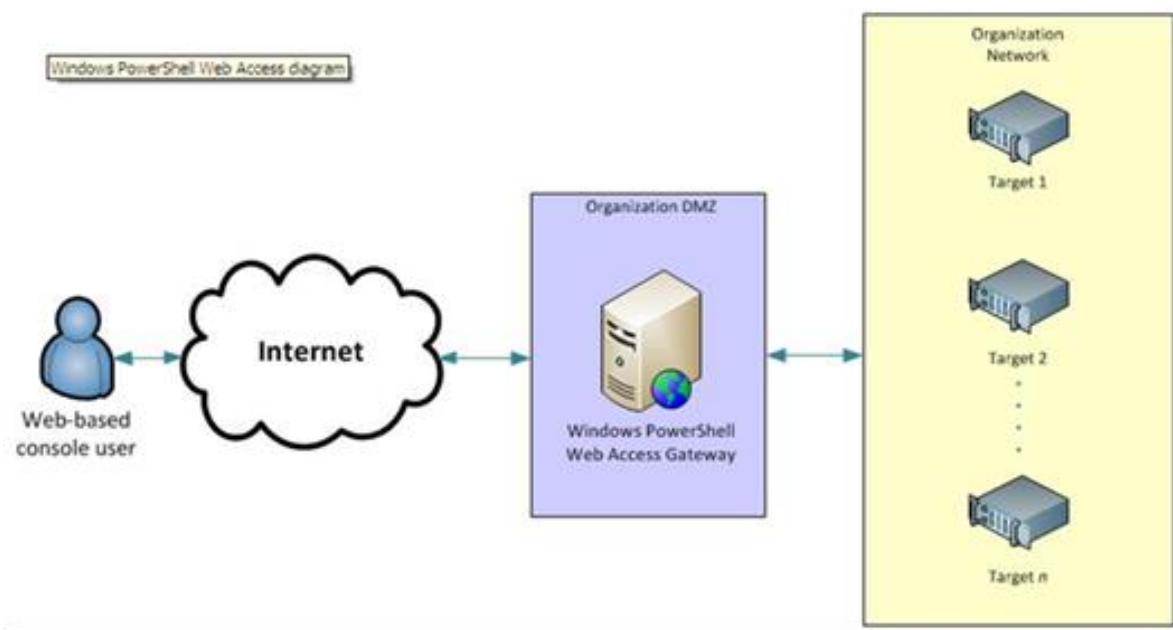
Při volání cmdletu Restart-Computer můžete určit, na jakou část operačního systému počkáte. V současné době jsou k dispozici tři možnosti: PowerShell, WinRM, WMI. Až bude příslušná část systému spuštěna, skript bude pokračovat v dalším běhu. Jedno z možných použití bude určitě ve workflow, kdy při složitějších operacích můžete jednoduše počkat na zkonfigurování počítače (či počítačů).

Jednoduché použití nových parametrů je vidět v aktuálním banneru Mistra Skriptíka.

PowerShell Web Access

Jedna ze skvělých novinek. Vzdálený přístup na server s PowerShellem pomocí prohlížeče. Vzhledem k použitým technologiím tedy možnost spravovat váš server například pomocí mobilního telefonu.

PowerShell Web Access (dále jen pswa) webová aplikace běžící na IIS serveru. Tato aplikace slouží jako gateway, přes kterou se můžete připojit na jakýkoli počítač (se zapnutým a konfigurovaným vzdáleným přístupem) ve vaší společnosti. Vše funguje následujícím způsobem (obrázek převzat z TechNetu, link na konci tohoto článku):



Jak pswa nakonfigurovat si ukážeme dále.

Nejprve je potřeba nainstalovat potřebnou část systému:

```
PS> Install-WindowsFeature -Name WindowsPowerShellWebAccess
```

Vše můžeme samozřejmě udělat i přes Server Manager. Jelikož pswa potřebuje pro svůj běh IIS, je potřeba jej mít nainstalovaný nebo doinstalovat současně s pswa.

Po instalaci je potřeba spustit instalaci vlastní aplikace pro pswa. Vzhledem k tomu, že celá komunikace probíhá přes HTTPS, potřebujeme pro správný běh SSL certifikát.

Malá odbočka: Ve všech návodech (včetně tohoto) se dočtete, že byste pro běh ve firemním prostředí neměli používat self-signed SSL certifikát. Vzhledem k tomu, že cmdlet určený k instalaci pswa má parametr, který takový certifikát vytváří, jedná se o nejjednodušší možnost vyzkoušení služby (při mém posledním zkoušení mi celá instalace trvala zhruba minutu). Ovšem je třeba mít na paměti, že pro reálný běh je opravdu dobré využít certifikát vydaný interní či externí certifikační autoritou.

```
PS> Install-PswaWebApplication -UseTestCertificate
Creating application pool pswa_pool...
```

Name	State	Applications
pswa_pool	Started	

```
Creating web application pswa...
```

```
Path : /pswa
ApplicationPool : pswa_pool
EnabledProtocols : http
PhysicalPath : c:\Windows\Web\PowerShellWebAccess\wwwroot
```

```
Creating self-signed certificate...
```

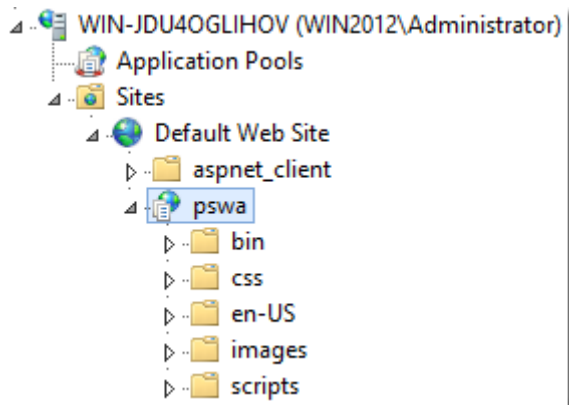
```
Create HTTPS Binding...
```

Po úspěšné instalaci je ještě potřeba přidat autorizační pravidlo. Tímto pravidlem určujeme, kdo se bude moci na náš server připojit.

```
PS> Add-PswaAuthorizationRule -UserName win2012\administrator -ComputerName win2012 -ConfigurationName Microsoft.PowerShell
```

Id	RuleName	User	Destination	ConfigurationName
0	Rule 0	win2012\administrator	win2012	Microsoft.PowerShell

Tím je instalace a konfigurace ukončena. Jak již bylo řečeno – pro testovací podmínky je jedná o opravdu jednoduché kroky. V IIS Manageru, pak pswa vypadá následovně:



Nyní se můžeme připojit přes prohlížeč na server, kde nám pswa běží. Vzhledem k tomu, že jsme použili self-signed certifikát, zobrazí se informace o „bezpečnostním problému“. Vzhledem k tomu, že jsme si jisti, můžeme potvrdit volbu „not recommended“ a pokračovat dále.

We recommend that you close this webpage and do not continue to this website.

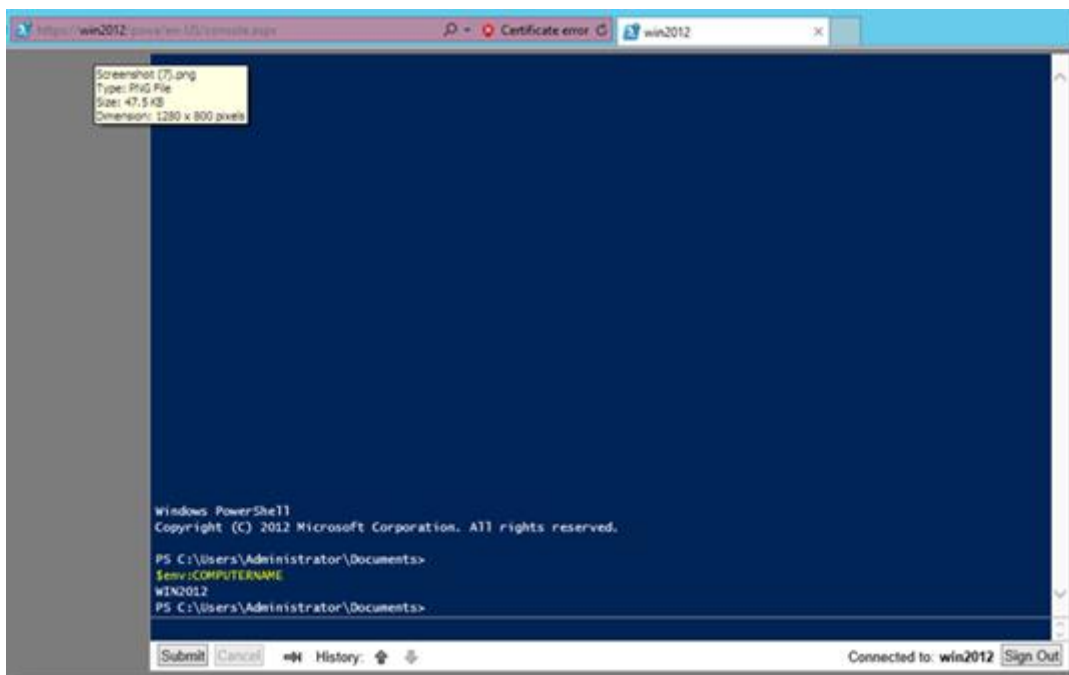
 [Click here to close this webpage.](#)

 [Continue to this website \(not recommended\).](#)

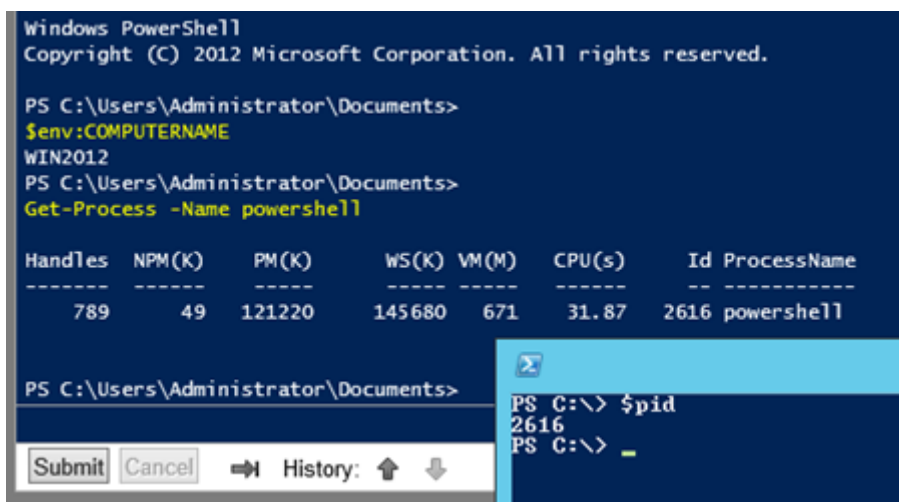
V dalším okně již vyplníme informace, které jsme definovali při tvorbě autorizačního pravidla a klikneme na Sign In.

Tímto přihlášením se nám otevírá celý nový svět (nebo minimálně dveře do naší firmy J

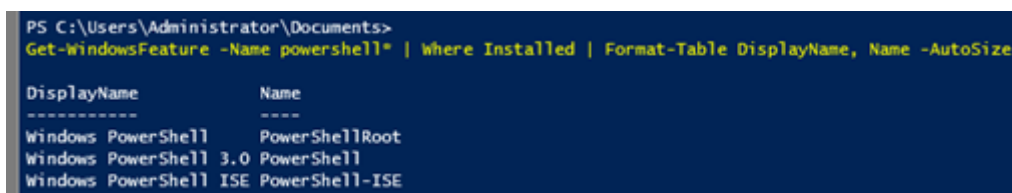
Vidíte, že okno PowerShellu běží ve webovém prohlížeči a funguje stejně, jako kdybychom seděli u konzole:



Funguje zde dokonce doplňování jmen cmdletů a parametrů pomocí tabulátoru. Jen pro kontrolu, na následujícím obrázku vidíte, že process PowerShellu spuštěný na serveru, kde běží IIS je viditelný i přes pswa:



Přes pswa můžete samozřejmě zkoušet i složitější propojení cmdletů pomocí roury:



Vše se chová naprosto stejně jako klasická konzole. Pokud máte na gateway serveru zpřístupněné moduly pro administraci vašich interních serverů, není opravdu problém připojit se odkudkoli a

spravovat jakýkoli zdroj ve vaší síti. Osobně si myslím, že pswa je jedna z nejlepších novinek PowerShellu v3 na Windows Serveru 2012.

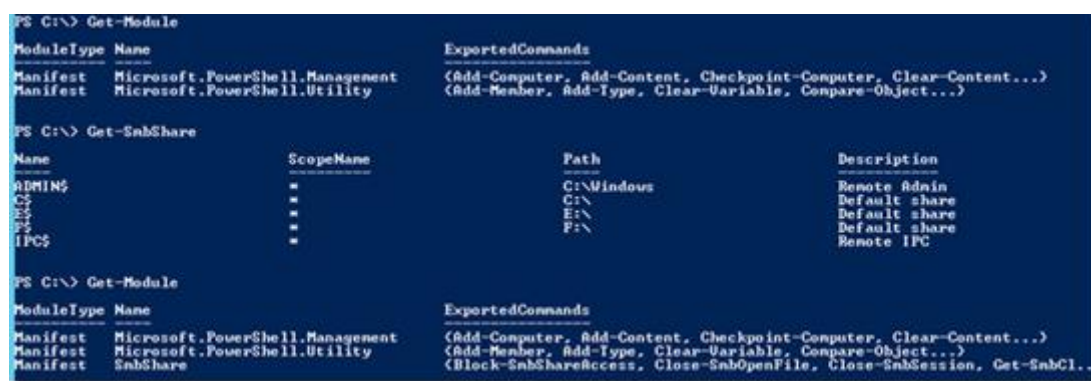
Samozřejmě, že pro firemní použití je potřeba upravit standardní nastavení, které jsem dnes ukázal, ale pro demonstraci použitelnosti toto určitě stačí.

Pokud by vás zajímaly opravdu detailní informace ke konfiguraci, můžete se podívat na článek [Deploy Windows PowerShell Web Access](#) na TechNetu.

JAK NA SDÍLENÉ SLOŽKY

Automatické načítání modulů

Než se pustíme do prohlídky dvou nových modulů, povíme si něco o možnosti takzvaného *module autoloading*. Pokud jste v PowerShell v2 potřebovali pracovat s cmdletem patřícím do určitého modulu, museli jste nejdříve tento modul načíst. V PowerShellu v3 již toto není potřeba. Stačí poprvé cmdlet použít a modul se automaticky importuje, jako kdybychom použili **Import-Module**. Podívejte se na následující obrázek:



```
PS C:\> Get-Module

ModuleType Name
-----
Manifest Microsoft.PowerShell.Management
Manifest Microsoft.PowerShell.Utility

PS C:\> Get-SmbShare

Name ScopeName Path Description
----
ADMIN$ = C:\Windows Remote Admin
C$ = C:\ Default share
E$ = E:\ Default share
F$ = F:\ Default share
IPC$ = Remote IPC

PS C:\> Get-Module

ModuleType Name
-----
Manifest Microsoft.PowerShell.Management
Manifest Microsoft.PowerShell.Utility
Manifest SmbShare
```

Při startu PowerShellu se načetly pouze dva základní moduly. Poté jsem rovnou zavolal cmdlet **Get-SmbShare**. Ihned poté je vidět ve výpisu opětovného volání **Get-Module**, že modul SmbShare se načetl automaticky.

Automatické načítání funguje i při použití tabulátoru (doplňování jmen cmdletů), což je na jednu stranu výhoda při hledání vhodného příkazu. Na druhou stranu se tím zvětšuje možnost dostupných cmdletů při automatickém doplňování. Pokud by se vám toto automatické načítání nelíbilo, lze jej vypnout pomocí proměnné **\$PSModuleAutoLoadingPreference**.

Sdílené složky

Dlouhou dobu jsme čekali na modul pro správu sdílených složek. Existovaly moduly vytvořené různými lidmi, ale oficiální podporu jsme neměli. S PowerShellem v3 přichází modul SmbShare, který v sobě obsahuje množství (přesně 28) užitečných cmdletů. Jelikož si myslím, že si zaslouží pozornost, budeme se dnes věnovat pouze tomuto modulu.

Nejdříve si ukážeme dostupné funkce (modul obsahuje opravdu funkce, nikoli cmdlety).

```
PS C:\> Get-Command -Module smbshare
```

CommandType	Name	ModuleName
-----	----	-----
Function	Block-SmbShareAccess	SmbShare
Function	Close-SmbOpenFile	SmbShare
Function	Close-SmbSession	SmbShare
Function	Get-SmbClientConfiguration	SmbShare
Function	Get-SmbClientNetworkInterface	SmbShare
Function	Get-SmbConnection	SmbShare
Function	Get-SmbMapping	SmbShare
Function	Get-SmbMultichannelConnection	SmbShare
Function	Get-SmbMultichannelConstraint	SmbShare
Function	Get-SmbOpenFile	SmbShare
Function	Get-SmbServerConfiguration	SmbShare
Function	Get-SmbServerNetworkInterface	SmbShare
Function	Get-SmbSession	SmbShare
Function	Get-SmbShare	SmbShare
Function	Get-SmbShareAccess	SmbShare
Function	Grant-SmbShareAccess	SmbShare
Function	New-SmbMapping	SmbShare
Function	New-SmbMultichannelConstraint	SmbShare
Function	New-SmbShare	SmbShare
Function	Remove-SmbMapping	SmbShare
Function	Remove-SmbMultichannelConstraint	SmbShare
Function	Remove-SmbShare	SmbShare
Function	Revoke-SmbShareAccess	SmbShare
Function	Set-SmbClientConfiguration	SmbShare
Function	Set-SmbServerConfiguration	SmbShare
Function	Set-SmbShare	SmbShare
Function	Unblock-SmbShareAccess	SmbShare
Function	Update-SmbMultichannelConnection	SmbShare

Je vidět, že si dal Microsoft opravdu záležet a z pohledu správce souborového serveru zde nechýbí nic důležitého. Na některé funkce se podíváme podrobněji.

Get-SmbShare

Seznam sdílených složek zobrazíme pomocí **Get-SmbShare**:

```
PS C:\> Get-SmbShare
```

Name	ScopeName	Path	Description
----	-----	----	-----
ADMIN\$	*	C:\Windows	Remote Admin
C\$	*	C:\	Default share
E\$	*	E:\	Default share
F\$	*	F:\	Default share
IPC\$	*		Remote IPC

Všechny příkazy spouštím na čisté instalaci Windows 2012 Serveru, takže nejsou dostupné jiné, než administrativní sdílené složky. Každá složka obsahuje některé další zajímavé vlastnosti:

```
PS C:\> Get-SmbShare -Name c$ | Format-List *
```

```
PresetPathAcl      :
ShareState         : Online
AvailabilityType    : NonClustered
ShareType          : FileSystemDirectory
FolderEnumerationMode : Unrestricted
CachingMode        : Manual
CATimeout          : 0
ConcurrentUserLimit : 0
ContinuouslyAvailable : False
CurrentUsers       : 0
Description        : Default share
EncryptData        : False
Name               : C$
Path               : C:\
Scoped             : False
ScopeName          : *
SecurityDescriptor  :
O:SYG:SYD:(A;;;GA;;;BA)(A;;;GA;;;BO)(A;;;GA;;;IU)
ShadowCopy         : False
Special            : True
Temporary          : False
Volume             : \\?\Volume{de1d5a93-11b8-11e2-93e8-806e6f6e6963}\
PSComputerName     :
CimClass           : ROOT/Microsoft/Windows/SMB:MSFT_SmbShare
CimInstanceProperties : {AvailabilityType, CachingMode, CATimeout,
ConcurrentUserLimit...}
CimSystemProperties :
Microsoft.Management.Infrastructure.CimSystemProperties
```

New-SmbShare

Pojďme vytvořit novou sdílenou složku. Schválně nebudu zadávat žádné parametry z příkazové řádky, aby bylo vidět, které jsou povinné.

```
PS C:\> New-SmbShare
```

```
cmdlet New-SmbShare at command pipeline position 1
Supply values for the following parameters:
Path: c:\temp\share1
Name: Share1
```

Name	ScopeName	Path	Description
----	-----	----	-----
Share1	*	c:\temp\share1	

Pokud se nyní podíváme na práva ke sdílení, jsou nastavena na Everyone/Read. To nám nemusí vždy vyhovovat (a asi opravdu nebude). Proto využijeme nastavení práv již při vytváření sdílení.

```
PS C:\> New-SmbShare -Path C:\Temp\Share2 -Name Share2 -FullAccess
Administrators -NoAccess Everyone
```

Name	ScopeName	Path	Description
----	-----	----	-----
Share2	*	C:\Temp\Share2	

New-SmbShare totiž obsahuje parametry **ChangeAccess**, **FullAccess**, **NoAccess** a **ReadAccess**, pomocí kterých můžeme přístup ovlivnit.

Z prvního výpisu je vidět, že práva na složky se zobrazují jako **SecurityDescriptor**

```
SecurityDescriptor : O:SYG:SYD:(A;;;GA;;;BA)(A;;;GA;;;BO)(A;;;GA;;;IU)
```

což není úplně příjemné počtení. Proto se nám bude hodit funkce **Get-SmbShareAccess**, která zobrazí práva v přehledné formě:

```
PS C:\> Get-SmbShareAccess Share1
```

Name	ScopeName	AccountName	AccessControlType	AccessRight
----	-----	-----	-----	-----
Share1	*	Everyone	Allow	Read

Set-SmbShare

Pokud již máme sdílený adresář vytvořený, můžeme měnit jeho nastavení pomocí **Set-SmbShare**. Jednou z možností, jak nastavit práva na Share1 podle Share2 je například toto.

```
PS C:\> Set-SmbShare -Name Share1 -SecurityDescriptor (Get-SmbShare
-Name Share2).SecurityDescriptor
```

Confirm

Are you sure you want to perform this action?Performing operation
'Modify' on Target '*,Share3'.

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):

Pokud bychom toto nastavení prováděli ze skriptu, je potřeba použít parametr **Force**, abychom se „nezasekli“ na potvrzovacím dialogu.

Remove-SmbShare

Životní cyklus našich sdílených adresářů ukončíme pomocí **Remove-SmbShare**.

```
PS C:\> Get-SmbShare share*
```

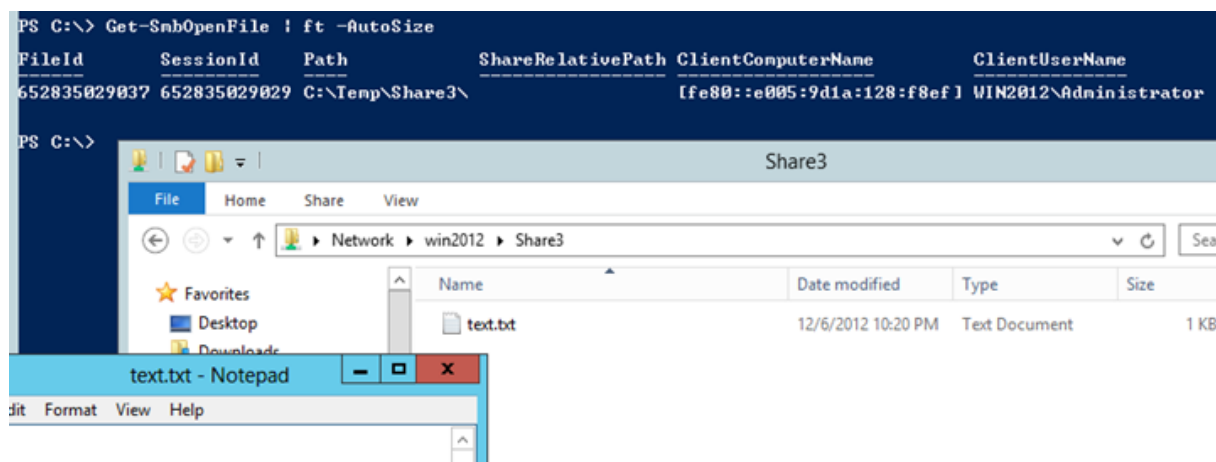
Name	ScopeName	Path	Description
----	-----	----	-----
Share1	*	c:\temp\share1	
Share2	*	C:\Temp\Share2	

```
PS C:\> Get-SmbShare share* | Remove-SmbShare -WhatIf
What if: Performing operation 'Remove-Share' on Target '*,Share1'.
What if: Performing operation 'Remove-Share' on Target '*,Share2'.
```

```
PS C:\> Get-SmbShare share* | Remove-SmbShare -Force
PS C:\> Get-SmbShare share*
```

*-SmbOpenFile

Na souborovém serveru se hodí, podívat se, které soubory jsou otevřené a kým.



Z obrázku je vidět, že jsem přes vytvořený share otevřel soubor *test.txt*. Poté jsem použil **Get-SmbOpenFile** pro zobrazení tohoto spojení.

```
PS C:\> Get-SmbOpenFile | Format-Table -AutoSize
```

FileId	SessionId	Path	ShareRelativePath
-----	-----	----	-----
652835029037	652835029029	C:\Temp\Share3\	[fe80::e005:9d1a:128:f8ef]
WIN2012\Administrator			

```
PS C:\> Close-SmbOpenFile
```

```
Confirm
Are you sure you want to perform this action?Performing operation
'Close-File' on Target '652835029037'.
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):Y
```

Otevřené spojení pak mohu i zavřít.

Další zajímavé funkce

Pro zjištění nastavení serveru slouží **Get-SmbServerConfiguration**.

```
PS C:\> Get-SmbServerConfiguration
```

```
AnnounceServer           : False
AsynchronousCredits       : 64
AutoShareServer           : True
AutoShareWorkstation      : True
CachedOpenLimit           : 5
AnnounceComment           :
EnableDownlevelTimewarp   : False
EnableLeasing             : True
EnableMultiChannel        : True
EnableStrictNameChecking  : True
AutoDisconnectTimeout     : 0
DurableHandleV2TimeoutInSeconds : 30
EnableAuthenticateUserSharing : False
EnableForcedLogoff        : True
EnableOplocks             : True
EnableSecuritySignature   : False
ServerHidden              : True
IrpStackSize              : 15
KeepAliveTime             : 2
MaxChannelPerSession      : 32
MaxMpxCount               : 50
MaxSessionPerConnection   : 16384
MaxThreadsPerQueue        : 20
MaxWorkItems              : 1
NullSessionPipes          :
NullSessionShares         :
OplockBreakWait           : 35
PendingClientTimeoutInSeconds : 120
RequireSecuritySignature  : False
EnableSMB1Protocol        : True
EnableSMB2Protocol        : True
Smb2CreditsMax            : 2048
Smb2CreditsMin            : 128
SmbServerNameHardeningLevel : 0
TreatHostAsStableStorage  : False
```

```

ValidateAliasNotCircular      : True
ValidateShareScope           : True
ValidateShareScopeNotAliased : True
ValidateTargetName           : True
EncryptData                  : False
RejectUnencryptedAccess      : True

```

V případě, že byste potřebovali některou hodnotu změnit, můžete použít naopak **Set-SmbServerConfiguration**.

Co třeba nastavení síťového rozhraní:

```
PS C:\> Get-SmbServerNetworkInterface
```

```

Scope Name Interface Index RSS Capable RDMA Capable Speed  IpAddress
-----
*      12      False   False   54 Mbps 192.168.10.101
*      12      False   False   54 Mbps fe80::e005:9d1a:...

```

Celý modul obsahuje funkce pro práci s následujícími vlastnostmi:

```
PS C:\> Get-Command -Module smbshare |% { $verb,$noun = $_.name -
split '-'; $noun} | group | sort count -desc
```

```

Count Name                                     Group
-----
      5 SmbShareAccess                        {SmbShareAccess, SmbShareAccess,
SmbShareAccess, SmbShareAccess...}
      4 SmbShare                              {SmbShare, SmbShare, SmbShare,
SmbShare}
      3 SmbMapping                            {SmbMapping, SmbMapping, SmbMapping}
      3 SmbMultichannelConstraint             {SmbMultichannelConstraint,
SmbMultichannelConstrain...}
      2 SmbOpenFile                          {SmbOpenFile, SmbOpenFile}
      2 SmbSession                           {SmbSession, SmbSession}
      2 SmbClientConfiguration               {SmbClientConfiguration,
SmbClientConfiguration}
      2 SmbMultichannelConnection            {SmbMultichannelConnection,
SmbMultichannelConnection}
      2 SmbServerConfiguration               {SmbServerConfiguration,
SmbServerConfiguration}
      1 SmbClientNetworkInterface            {SmbClientNetworkInterface}
      1 SmbConnection                        {SmbConnection}
      1 SmbServerNetworkInterface            {SmbServerNetworkInterface}

```

Pokud by vás zajímala nápověda k dalším funkcím, najdete ji na TechNetu v sekci [SMB Share Cmdlets in Windows PowerShell](#).

PRÁCE SE SÍŤOVÝM ADAPTÉREM

Pokud se učíte jakýkoli nový program, je vždy nejlepší používat jej co nejčastěji. To samé platí i pro PowerShell. Proto jsem již hodně dlouho nepustil cmd.exe a vše spouštím právě z konzole PowerShellu. Dalším krokem je nahrazení nativních příkazů Windows jejich cmdlety a funkcemi. Typickým případem jsou dva nejčastější: ping a ipconfig. Ping má svůj protějšek v cmdletu **Test-Connection** a ipconfig ve funkci **Get-NetIPConfiguration**. Dnes si povíme právě o funkcích sloužících k práci se síťovým rozhraním.

Modul NetTCPIP

Ukažme si dostupné funkce.

```
PS C:\> Get-Command -Module NetTCPIP
```

CommandType	Name	ModuleName
-----	----	-----
Function	Get-NetIPAddress	NetTCPIP
Function	Get-NetIPConfiguration	NetTCPIP
Function	Get-NetIPInterface	NetTCPIP
Function	Get-NetIPv4Protocol	NetTCPIP
Function	Get-NetIPv6Protocol	NetTCPIP
Function	Get-NetNeighbor	NetTCPIP
Function	Get-NetOffloadGlobalSetting	NetTCPIP
Function	Get-NetPrefixPolicy	NetTCPIP
Function	Get-NetRoute	NetTCPIP
Function	Get-NetTCPConnection	NetTCPIP
Function	Get-NetTCPSetting	NetTCPIP
Function	Get-NetTransportFilter	NetTCPIP
Function	Get-NetUDPEndpoint	NetTCPIP
Function	Get-NetUDPSetting	NetTCPIP
Function	New-NetIPAddress	NetTCPIP
Function	New-NetNeighbor	NetTCPIP
Function	New-NetRoute	NetTCPIP
Function	New-NetTransportFilter	NetTCPIP
Function	Remove-NetIPAddress	NetTCPIP
Function	Remove-NetNeighbor	NetTCPIP
Function	Remove-NetRoute	NetTCPIP
Function	Remove-NetTransportFilter	NetTCPIP
Function	Set-NetIPAddress	NetTCPIP
Function	Set-NetIPInterface	NetTCPIP
Function	Set-NetIPv4Protocol	NetTCPIP
Function	Set-NetIPv6Protocol	NetTCPIP
Function	Set-NetNeighbor	NetTCPIP
Function	Set-NetOffloadGlobalSetting	NetTCPIP
Function	Set-NetRoute	NetTCPIP
Function	Set-NetTCPSetting	NetTCPIP
Function	Set-NetUDPSetting	NetTCPIP

Stejně jako v [minulém díle](#) našeho seriálu vidíme, že množství funkcí je opravdu velké.

Get-NetIPConfiguration

Podíváme se rovnou na náhradu příkazu ipconfig:

```
PS C:\> Get-NetIPConfiguration
```

```
InterfaceAlias      : Wi-Fi
InterfaceIndex      : 12
InterfaceDescription : Intel(R) WiFi Link
NetProfile.Name      : WiFi
IPv4Address          : 192.168.100.10
IPv6DefaultGateway  :
IPv4DefaultGateway  : 192.168.100.1
DNSServer            : 192.168.100.1

InterfaceAlias      : Ethernet
InterfaceIndex      : 13
InterfaceDescription : Intel(R) Gigabit Network Connection
NetAdapter.Status    : Disconnected
```

Z výstupu je jasné, že aktivní připojení je přes Wi-Fi a že máme dále dostupný ještě síťový adaptér Ethernet, který je momentálně odpojený. Pokud bychom použili parametr **Detailed**, dozvíme se podrobnější informace (podobné použití ipconfig /all).

Na tomto místě upozorním ještě na jeden chyták. Už jste se určitě potkali se situací, kdy jste si všimli, že ne všechny vlastnosti určitého zdroje (soubor, process, ...) jsou ve výstupu vidět. Stejně je to i nyní. Pokud chcete zobrazit opravdu všechny parametry vašeho připojení, zkuste použít cmdlet **Format-List**:

```
PS C:\> Get-NetIPConfiguration | Format-List *
```

```
ComputerName        : WIN8
InterfaceAlias       : Wi-Fi
InterfaceIndex       : 12
InterfaceDescription : Intel(R) WiFi Link AGN
NetAdapter           : MSFT_NetAdapter (CreationClassName =
"MSFT_NetAdapter", DeviceID = "{0-5EF4-449A-AFB5-2E082768F5}",
SystemCreationClassName = "CIM_NetworkPort", SystemName =
"Win8")
NetIPv6Interface     : MSFT_NetIPInterface (Name =
";?55??55;", CreationClassName = "", SystemCreationClassName =
"", SystemName = "")

NetIPv4Interface     : MSFT_NetIPInterface (Name = ";?55?55;",
CreationClassName = "", SystemCreationClassName = "",
SystemName = "")
NetProfile            : MSFT_NetConnectionProfile (InstanceID =
```

```
"{F8BE4A34-5EF4-449A-AFB5-2EC8282768F5}")
AllIPAddresses      : {192.168.100.103,
ab80::c271:b947:89c0:bdea%12}
IPv6Address         : {}
IPv6TemporaryAddress : {}
IPv6LinkLocalAddress : {ab80::c271:b947:89c0:bdea%12}
IPv4Address         : {192.168.100.103}
IPv6DefaultGateway  :
IPv4DefaultGateway  : {MSFT_NetRoute (InstanceID =
":8:8:8:9:55;?55;C?8;@B8;:8;55;")}
DNSServer           : {MSFT_DNSClientServerAddress (Name =
"12", CreationClassName = "", SystemCreationClassName = "",
SystemName = "23"), MSFT_DNSClientServerAddress (Name = "12",
CreationClassName = "", SystemCreationClassName = "",
SystemName = "2")}
Detailed            : False
```

Get-NetIPInterface

Další možností je samozřejmost podívat se na konkrétní síťové rozhraní

```
PS C:\> Get-NetIPInterface
```

ifIndex	InterfaceAlias	AddressFamily	NlMtu(Bytes)	InterfaceMetric	Dhcp	ConnectionState	PolicyStore
13	Ethernet	IPv6	1500	5	Disabled	Disconnected	ActiveStore
15	Teredo Tunneling	IPv6	1280	50	Disabled	Connected	ActiveStore
14	isatap.{F8-449A-A...}	IPv6	1280	50	Disabled	Disconnected	ActiveStore
12	Wi-Fi	IPv6	1500	25	Enabled	Connected	ActiveStore
1	Loopback	IPv6	4294967295	50	Pseudo-Disabled	Connected	ActiveStore
13	Ethernet	IPv4	1500	5	Enabled	Disconnected	ActiveStore
12	Wi-Fi	IPv4	1500	25	Enabled	Connected	ActiveStore
1	Loopback	IPv4	4294967295	50	Pseudo-Disabled	Connected	ActiveStore

Ani u této funkce nezapomeňte použít **Format-List!**

NetAdapter

Druhým užitečným modulem pro práci se sítí je NetAdapter. Jak je již z názvu patrné, pracuje přímo s adaptérem. Pojdme si opět ukázat dostupné funkce:

```
PS C:\> Get-Command -Module NetAdapter
```

CommandType	Name	ModuleName
Function	Disable-NetAdapter	NetAdapter
Function	Disable-NetAdapterBinding	NetAdapter
Function	Disable-NetAdapterChecksumOffload	NetAdapter
Function	Disable-NetAdapterEncapsulatedPacketTaskOffload	NetAdapter
Function	Disable-NetAdapterIPsecOffload	NetAdapter
Function	Disable-NetAdapterLso	NetAdapter
Function	Disable-NetAdapterPowerManagement	NetAdapter
Function	Disable-NetAdapterQos	NetAdapter
Function	Disable-NetAdapterRdma	NetAdapter
Function	Disable-NetAdapterRsc	NetAdapter
Function	Disable-NetAdapterRss	NetAdapter
Function	Disable-NetAdapterSriov	NetAdapter
Function	Disable-NetAdapterVmq	NetAdapter
Function	Enable-NetAdapter	NetAdapter
Function	Enable-NetAdapterBinding	NetAdapter
Function	Enable-NetAdapterChecksumOffload	NetAdapter
Function	Enable-NetAdapterEncapsulatedPacketTaskOffload	NetAdapter
Function	Enable-NetAdapterIPsecOffload	NetAdapter
Function	Enable-	

NetAdapterLso	NetAdapter
Function Enable-	
NetAdapterPowerManagement	NetAdapter
Function Enable-	
NetAdapterQos	NetAdapter
Function Enable-	
NetAdapterRdma	NetAdapter
Function Enable-	
NetAdapterRsc	NetAdapter
Function Enable-	
NetAdapterRss	NetAdapter
Function Enable-	
NetAdapterSriov	NetAdapter
Function Enable-	
NetAdapterVmq	NetAdapter
Function Get-	
NetAdapter	NetAdapter
Function Get-	
NetAdapterAdvancedProperty	NetAdapter
Function Get-	
NetAdapterBinding	NetAdapter
Function Get-	
NetAdapterChecksumOffload	NetAdapter
Function Get-	
NetAdapterEncapsulatedPacketTaskOffload	NetAdapter
Function Get-	
NetAdapterHardwareInfo	NetAdapter
Function Get-	
NetAdapterIPsecOffload	NetAdapter
Function Get-	
NetAdapterLso	NetAdapter
Function Get-	
NetAdapterPowerManagement	NetAdapter
Function Get-	
NetAdapterQos	NetAdapter
Function Get-	
NetAdapterRdma	NetAdapter
Function Get-	
NetAdapterRsc	NetAdapter
Function Get-	
NetAdapterRss	NetAdapter
Function Get-	
NetAdapterSriov	NetAdapter
Function Get-	
NetAdapterSriovVf	NetAdapter
Function Get-	
NetAdapterStatistics	NetAdapter
Function Get-	
NetAdapterVmq	NetAdapter
Function Get-	
NetAdapterVmqQueue	NetAdapter

Function	Get-	
NetAdapterVPort		NetAdapter
Function	New-	
NetAdapterAdvancedProperty		NetAdapter
Function	Remove-	
NetAdapterAdvancedProperty		NetAdapter
Function	Rename-	
NetAdapter		NetAdapter
Function	Reset-	
NetAdapterAdvancedProperty		NetAdapter
Function	Restart-	
NetAdapter		NetAdapter
Function	Set-	
NetAdapter		NetAdapter
Function	Set-	
NetAdapterAdvancedProperty		NetAdapter
Function	Set-	
NetAdapterBinding		NetAdapter
Function	Set-	
NetAdapterChecksumOffload		NetAdapter
Function	Set-	
NetAdapterEncapsulatedPacketTaskOffload		NetAdapter
Function	Set-	
NetAdapterIPsecOffload		NetAdapter
Function	Set-	
NetAdapterLso		NetAdapter
Function	Set-	
NetAdapterPowerManagement		NetAdapter
Function	Set-	
NetAdapterQos		NetAdapter
Function	Set-	
NetAdapterRdma		NetAdapter
Function	Set-	
NetAdapterRsc		NetAdapter
Function	Set-	
NetAdapterRss		NetAdapter
Function	Set-	
NetAdapterSriov		NetAdapter
Function	Set-	
NetAdapterVmq		NetAdapter

Poznámka: Rád bych upozornil na jednu věc. Vzhledem k množství dostupných funkcí, není mým cílem dnes ukázat jejich výstupy. Jde mi spíše o ukázání cesty a pouze některých (nejzajímavějších?). Jak jsem již řekl v prvním odstavci – PowerShell se nejlépe naučíte tak, že jej budete stále používat. Vyberte si proto funkci, která vám přijde zajímavá (ze začátku některou z **Get-***), a podívejte se na její výstup. Každý den pak můžete přidat další.

Get-NetAdapter

Zkuste prozkoumat tuto funkci. Jak z ní získáte co nejvíc informací? Mohli byste postupovat například takto:

```
PS C:\> Get-NetAdapter
PS C:\> Get-Help Get-NetAdapter
PS C:\> Get-NetAdapter -Name Wi-Fi
PS C:\> Get-NetAdapter -Name Wi-Fi | Format-List *
PS C:\> Get-NetAdapter -Name Wi-Fi | Get-Member
```

Výstupem cmdletu **Get-Member** je objekt typu:

- Microsoft.Management.Infrastructure.CimInstance#ROOT/StandardCimv2/MSFT_NetAdapter

Podívejte se na [MSFT_NetAdapter class](#) článek na MSDN – stále je se co učit J

A co třeba **Get-NetAdapterAdvancedProperty**?

VYLEPŠENÁ PRÁCE S WMI

Práce s WMI (Windows Management Instrumentation) je v PowerShellu vynikající již od verze 1. Pro mne osobně byl pádným argumentem pro přechod na PowerShell cmdlet **Get-WmiObject**. S jeho pomocí jsem mohl (a samozřejmě stále mohu) přistupovat ke svému ConfigMgr serveru. **Get-WmiObject** (a jeho alias **gwmi**) vám zpřístupňuje WMI velice jednoduchým způsobem. Ve spolupráci s **Get-Member** se často obejdete bez externí nápovědy.

Malá rozcvička pro ty, kteří **Get-WmiObject** nepoužívají:

```
PS C:\> Get-WmiObject -Class Win32_B* -List
```

Namespace: ROOT\cimv2

Name	Methods	Properties
----	-----	-----
Win32_BaseBoard	{IsCompatible}	{Caption, ConfigOptions,
CreationClassName, Depth...}		
Win32_BIOS	{}	{BiosCharacteristics,
BIOSVersion, BuildNumber, Caption...}		
Win32_BaseService	{StartService, St...	{AcceptPause, AcceptStop,
Caption, CreationClassName...}		
Win32_Battery	{SetPowerState, R...	{Availability,
BatteryRechargeTime, BatteryStatus, Caption...}		
Win32_Bus	{SetPowerState, R...	{Availability, BusNum, BusType,
Caption...}		
Win32_BootConfiguration	{}	{BootDirectory, Caption,
ConfigurationPath, Description...}		
Win32_Binary	{}	{Caption, Data, Description,
Name...}		
Win32_BindImageAction	{Invoke}	{ActionID, Caption, Description,
Direction...}		

Pomocí parametru **List** můžeme zjistit všechny třídy začínající na *Win32_B*. Ve výpisu nás zajímá třída Win32_BIOS a proto si zobrazíme její obsah:

```
PS C:\> Get-WmiObject -Class Win32_BIOS
```

```
SMBIOSBIOSVersion : A37
Manufacturer      : Dell Inc.
Name              : Phoenix ROM BIOS PLUS Version 1.10 A37
SerialNumber      : 20XXXX0
Version           : DELL - 1234abcd
```

Nezapomeňte, že některé objekty nezobrazují standardně všechny své vlastnosti. Proto i u předchozího příkladu, zkuste použít

```
PS C:\> Get-WmiObject -Class Win32_BIOS | Format-List *
```

V PowerShell v2 byly uvedeny některé nové cmdlety pro práci s WMI a ve verzi 3 jsme se dočkali další velké změny.

WMI vs. CIM

Aniž bych chtěl zabíhat do přílišných detailů, bylo by dobré zmínit, že WMI je Microsoft implementace standardu CIM (Common Information Model). V současnosti tedy máme v PowerShellu dvě možnosti přístupu do WMI:

Poznámka: Budu zde tyto dvě možnosti nazývat jako „WMI cmdlety“ a „CIM cmdlety“. Vždy v uvozovkách – nejedná se o žádný zavedený název, pouze chci odlišit „starý“ a „nový“ přístup.

„WMI cmdlety“ – Některé z nich jsou dostupné již od v1. Stále existují i ve verzi 3, ale vypadá to, že Microsoft je již dále nebude rozvíjet. Pro účely zpětné kompatibility, ale samozřejmě zůstávají (otázkou je, jak to bude v další verzi PowerShellu). Nevýhodou těchto cmdletů je, že potřebují pro své fungování RPC. Hodně problémů při práci s těmito cmdlety je způsobeno právě špatným nastavením RPC. Jste pak nuceni strávit svůj čas odhalováním potenciálního problému.

Seznam „WMI cmdletů“ v PowerShellu v3:

```
PS C:\> Get-Command -CommandType cmdlet -Name *wmi*
```

CommandType	Name	ModuleName
Cmdlet	Get-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Invoke-WmiMethod	Microsoft.PowerShell.Management
Cmdlet	Register-WmiEvent	Microsoft.PowerShell.Management
Cmdlet	Remove-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Set-WmiInstance	Microsoft.PowerShell.Management

„CIM cmdlety“ – S uvedením Windows Serveru 2012 jako cloud OS Microsoft přidal tyto cmdlety. Jejich výhodou je, že jsou schopny pracovat oproti WMI, ale i oproti jiným systémům využívajícím implementaci CIM standardu. Výsledkem je poté možnost spravovat například operační systémy Linux (pokud vás zajímá tato možnost více, zkuste si najít odkazy na NanoWBEM).

Další výhodou „CIM cmdletů“ je to, že komunikují pomocí protokolu WSMAN. Tento protokol využívá pro přenos dat mezi počítači protokol http (nebo HTTPS – v závislosti na vašem nastavení) a je tedy velice jednoduché jej nastavovat. WSMAN se používá již od PowerShellu v2 (kde byl použit poprvé pro vzdálený přístup). Vzhledem k tomu, že WSMAN je na serverech (od verze 2008 R2) standardně zapnut a nakonfigurován, je použití „CIM cmdletů“ transparentní.

Seznam „CIM cmdletů“ v PowerShellu v3:

```
PS C:\> Get-Command -CommandType cmdlet -Name *cim*
```

CommandType	Name	ModuleName
Cmdlet	Get-CimAssociatedInstance	CimCmdlets
Cmdlet	Get-CimClass	CimCmdlets
Cmdlet	Get-CimInstance	CimCmdlets

Cmdlet	Get-CimSession	CimCmdlets
Cmdlet	Invoke-CimMethod	CimCmdlets
Cmdlet	New-CimInstance	CimCmdlets
Cmdlet	New-CimSession	CimCmdlets
Cmdlet	New-CimSessionOption	CimCmdlets
Cmdlet	Register-CimIndicationEvent	CimCmdlets
Cmdlet	Remove-CimInstance	CimCmdlets
Cmdlet	Remove-CimSession	CimCmdlets
Cmdlet	Set-CimInstance	CimCmdlets

Pokud se podíváte na oba výpisy, najdete některé z cmdletů v obou z nich (i když s mírně odlišnými jmény):

- Get-WmiObject -> Get-CimInstance
- Invoke-WmiMethod -> Invoke-CimMethod
- Register-WmiEvent -> Register-CimIndicationEvent
- Remove-WmiObject -> Remove-CimInstance
- Set-WmiInstance -> Set-CimInstance

Pokud máte správně nastaveny protokoly RPC i WSMAN je jedno, který z příkazů použijete. Osobně doporučuji začít si zvykat na „CIM cmdlety“. Ještě jednou připomínám, že na Windows OS je jedno (z hlediska vrácených dat), který z cmdletů použijete.

Poznámka: Pro mne osobně je „těžké“ si na nové zvyknout. Vše je ale spíše způsobeno mými prsty, které automaticky stále píší **gwmi** místo hlavou chtěného **gcim** (alias pro [Get-CimInstance](#)).

Pojďme si porovnat výstupy **Get-WmiObject** a **Get-CimInstance**.

```
PS C:\> Get-CimInstance win32_bios
```

```
SMBIOSBIOSVersion : A06
Manufacturer       : Dell Inc.
Name               : Phoenix ROM BIOS PLUS Version 1.10 A06
SerialNumber       : 1234567
Version            : DELL - 12345abc
```

```
PS C:\> Get-WmiObject win32_bios
```

```
SMBIOSBIOSVersion : A06
Manufacturer       : Dell Inc.
Name               : Phoenix ROM BIOS PLUS Version 1.10 A06
SerialNumber       : 1234567
Version            : DELL - 12345abc
```

Vidíte, že výsledek je naprosto stejný. Můžeme si potvrdit i pomocí **Get-Member**, že oba cmdlety vrací stejnou třídu z WMI:

```
PS C:\> Get-WmiObject win32_bios | Get-Member
```

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_BIOS
```

```
PS C:\> Get-CimInstance win32_bios | Get-Member
```

```
TypeName:
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_BIOS
```

Jak bylo řečeno – vrácená třída je stejná (namespace root/cimv2), pouze typ objektu se liší v závislosti na použité metodě. Pro nás jako „koncové uživatele“ PowerShellu je jedno, jaká metoda je použita pod povrchem.

I když to na první pohled vypadá, že oba cmdlety jsou identické, jsou zde mírné změny v jejich syntaxi. Nalezení rozdílů ponechám plně na vašem samostudiu. Pouze upozorním na zřejmě největší

rozdíl. **Get-CimInstance** nemá (narodil od **Get-WmiObject**) parametr **Credential**. Pokud potřebujete spustit **gcim** na vzdáleném počítači s jiným jménem/heslem, musíte jej použít v kombinaci s **Invoke-Command**, například takto:

```
PS C:\> Invoke-Command -ScriptBlock { Get-CimInstance -Class Win32_BIOS } -  
Computer $server -Cred $cred
```

Po chvíli používání zjistíte, že „CIM cmdlety“ jsou trošku „ukecanější“.

Pojďme si ještě ukázat rozdíl mezi **Invoke-WmiMethod** a **Invoke-CimMethod**. Nejlépe na spuštění nějakého procesu.

```
PS C:\> Invoke-WmiMethod -Class Win32_Process -Name Create -ArgumentList  
notepad.exe
```

```
__GENUS          : 2  
__CLASS          : __PARAMETERS  
__SUPERCLASS     :  
__DYNASTY        : __PARAMETERS  
__RELPATH        :  
__PROPERTY_COUNT : 2  
__DERIVATION     : {}  
__SERVER         :  
__NAMESPACE     :  
__PATH           :  
ProcessId        : 4336  
ReturnValue       : 0
```

Výsledkem je spuštění notepadu. Ve výpisu vidíme, že **ReturnValue** je 0, tedy úspěch. Bohužel jsem před pár dny řešil problém, kdy spuštění procesu na vzdáleném počítači vracelo chybu 3. Po chvíli pátrání se ukázalo, že chyba byla v nastavení RPC. Pokud bych mohl použít PowerShell v3, měl bych práci o hodně ulehčenou:

```
PS C:\> Invoke-CimMethod -ClassName Win32_Process -Name Create -Arguments  
@{CommandLine='notepad.exe'}
```

```
ProcessId ReturnVal PSComputerName  
-----  
3720      0
```

Opět – „ukecanější“ zadání, jednodušší správa a možnost spuštění na vzdáleném počítači. Všimněte si parametru **Arguments**. Jeho hodnotou je hash tabulka. Výhodou je, že pokud volaná metoda (v našem případě **Create**) potřebuje více parametrů, můžete je zadat jednoduchým způsobem do této tabulky. V případě **Invoke-WmiMethod** musíte zapsat všechny parametry ve správném pořadí, což občas přináší nečekané strasti. Více například v článku od Shaye Levyho:

<http://blogs.microsoft.co.il/blogs/scriptfanatic/archive/2010/12/16/invoking-wmi-methods-in-powershell.aspx>

Pozor: Shay ukazuje příklad na formátování disku. Nepodlehňte pokušení zkoušet jeho příklad na počítači, na kterém vám záleží!

Pokud byste se chtěli o „nových CIM cmdletech“ dozvědět více, doporučuji podívat se na následující zdroje:

- <http://blogs.msdn.com/b/powershell/archive/2012/08/24/introduction-to-cim-cmdlets.aspx>
- <http://technet.microsoft.com/en-us/library/jj553783.aspx>

Pokud máte nainstalován PowerShell v3, doporučuji vám začít postupně používat „CIM cmdlety“. Vzhledem k tomu, že za nějakou dobu budou hlavní možností správy, vložený čas se vám rozhodně vrátí.

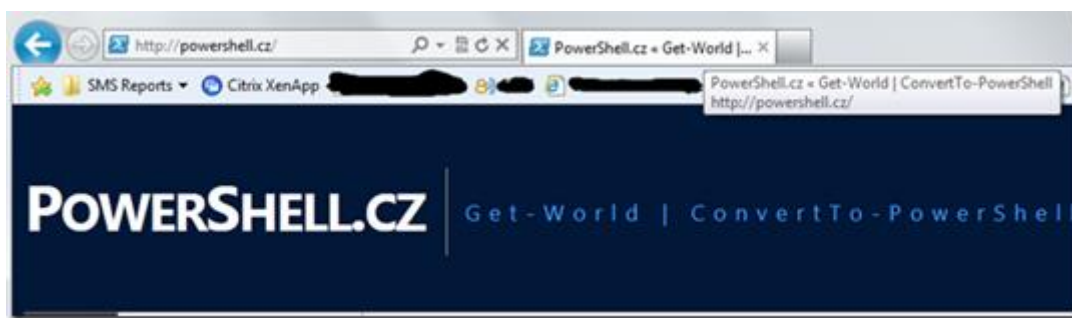
JAK NA WEBOVÉ STRÁNKY

Velmi často se stane, že potřebujete pracovat s webovými stránkami. At' se jedná o stažení celé stránky nebo zpracování její části (třeba na základě daného pravidla – pouze obrázky).

V PowerShellu v2 bylo potřeba použít .NET třídu [Net.WebClient](#). Například:

```
PS C:\> $w = New-Object Net.WebClient
PS C:\> $page = $w.DownloadString('http://www.powershell.cz')
PS C:\> $page -match '\<title\>(?\<Title>.*?)\</title\>'
PS C:\> $matches.Title
PowerShell.cz « Get-World | ConvertTo-PowerShell
```

Titulek stránky můžete potvrdit pohledem na vlastní stránku.



Nejprve vytvoříme proměnnou pro náš „webový“ objekt. Poté použijeme jeho metodu **DownloadString** a text stránky uložíme do proměnné *page*. V dalším kroku si pomocí regulárního výrazu zjistíme text titulku stránky, který si v další řádce vypíšeme.

Stažení stránky je tedy velmi jednoduché. Pro složitější operace je potřeba použít např. regulární výrazy. Což už pro některé administrátory může být nelehkou překážkou. Jak je vidět ze zápisu '`\<title\>(?\<Title>.*?)\</title\>`' nejedná se o žádnou oddechovku. Regulárními výrazy se zabývat nebudeme a zkusíme si ještě zjednodušit stažení stránky.

Pojďme tedy tedy přejít na PowerShell v3.

Invoke-WebRequest

Pojďme si tedy předchozí příklad přepsat:

```
PC C:\> $www = Invoke-WebRequest -Uri 'http://www.powershell.cz'
PC C:\> $www.ParsedHtml.title
PowerShell.cz « Get-World | ConvertTo-PowerShell
```

Velice jednoduché a efektivní řešení. Pojďme se blíže podívat na proměnnou *www*.

```
PS C:\> $www | Get-Member
```

TypeName: Microsoft.PowerShell.Commands.HtmlWebResponseObject

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
AllElements	Property	Microsoft.PowerShell.Commands.WebCmdletElementCollection AllElements {get;}

BaseResponse	Property	System.Net.WebResponse BaseResponse {get;set;}
Content	Property	string Content {get;}
Forms	Property	Microsoft.PowerShell.Commands.FormObjectCollection Forms {get;}
Headers	Property	System.Collections.Generic.Dictionary[string,string] Headers {get;}
Images	Property	Microsoft.PowerShell.Commands.WebCmdletElementCollection Images {get;}
InputFields	Property	Microsoft.PowerShell.Commands.WebCmdletElementCollection InputFields {get;}
Links	Property	Microsoft.PowerShell.Commands.WebCmdletElementCollection Links {get;}
ParsedHtml	Property	mshtml.IHTMLDocument2 ParsedHtml {get;}
RawContent	Property	string RawContent {get;}
RawContentLength	Property	long RawContentLength {get;}
RawContentStream	Property	System.IO.MemoryStream RawContentStream {get;}
Scripts	Property	Microsoft.PowerShell.Commands.WebCmdletElementCollection Scripts {get;}
StatusCode	Property	int StatusCode {get;}
StatusDescription	Property	string StatusDescription {get;}

Dalšími zajímavým vlastnostmi jsou například *Images* nebo *Links*.

```
PS C:\> $www.Images | Format-Table tagName, title, src -Auto
```

tagName	title	src
IMG	AddTFS	http://powershell.cz/wp-content/uploads/AddTFS-64x64.png
IMG		http://powershell.cz/wp-content/uploads/Includes/images/smilies/icon_smile.gif
IMG		http://powershell.cz/wp-content/uploads/AddTFS.png
IMG		http://powershell.cz/wp-content/uploads/tfslogin-277x300.png
IMG		http://powershell.cz/wp-content/uploads/tfsselect.png
IMG		http://powershell.cz/wp-content/uploads/tfsselect2.png
IMG		http://powershell.cz/wp-content/uploads/tfsOpen.png
IMG		http://powershell.cz/wp-content/uploads/tfschoosefile.png
IMG	FailFastConsole	http://powershell.cz/wp-content/uploads/FailFastConsole-64x64.png
IMG	FailFastConsole	http://powershell.cz/wp-content/uploads/FailFastConsole-150x150.png
IMG	FailFastWarningBox	http://powershell.cz/wp-content/uploads/FailFastWarningBox-150x150.png
IMG	FailFastEvent	http://powershell.cz/wp-content/uploads/FailFastEvent-150x150.png
IMG		http://powershell.cz/wp-content/uploads/Includes/images/smilies/icon_smile.gif

```
PS C:\> $www.Links | Format-Table title, href -Auto
```

title	href
Follow me on Twitter	https://twitter.com/Makovec

```
RSS Feeds          http://powershell.cz/?feed=rss2
You are Home       http://powershell.cz
about_PowerShell.cz http://powershell.cz/about/
(...)
```

Použitím přístupu k webu (nezávisle na použité metodě), můžeme vytvářet velice zajímavé funkce. Jednu z nich mám ve svém PowerShell profilu – kurzy zahraničních měn. Vzhledem k tomu, že na pracovním počítači mám PowerShell v3 od minulého týdne, zkusím přepsat současnou verzi (s použitím Net.WebClient) za pomoci **Invoke-WebRequest**.

Nejprve potřebujeme adresu, na které jsou aktuální kurzy uloženy. Jako ideální se jeví ČNB:

```
PS C:\> $url =
'http://www.cnb.cz/cs/financni_trhy/devizovy_trh/kurzy_devizoveho_trhu/denni_k
urz.txt'
```

Poté si pomocí cmdletu stáhneme data do proměnné – asi bychom to v tomto případě dělat nemuseli a mohli bychom rovnou použít rouru pro další zpracování. Nicméně, když používám funkce, mám rád jednotlivé části oddělené.

```
PS C:\> Invoke-WebRequest -Uri $url
```

Pokud budete tento cmdlet používat častěji, budete možná používat jeho alias **iwr**. Dále už jenom zpracujeme stažená data. Využijeme cmdlet **Select-Object** – od verze 3 obsahuje nový parameter, **Skip**. Tím určujeme, kolik prvků od začátku kolekce chceme přeskočit. Vzhledem k tomu, že struktura staženého souboru je následující:

```
PS C:\> $kurzy.Content
01.03.2013 #43
země|měna|množství|kód|kurz
Austrálie|dolar|1|AUD|20,153
Brazílie|real|1|BRL|9,944
Bulharsko|lev|1|BGN|13,129
(...)
```

Přeskočíme první dvě řádky a vytvoříme si vlastní popisky jednotlivých sloupců. Veškerou práci již nyní provedeme na jedné řádce.

```
PS C:\> $kurzy.Content -split "`n" | Select-Object -Skip 2 | `
    ConvertFrom-Csv -Delimiter '|' -Header 'Zeme','Mena','Mnozstvi','Kod','Kurz'
```

Nejprve se ujistíme, že každá řádka bude obsahovat jeden záznam. Poté přeskočíme první dva z nich (datum a hlavičku) a ze zbývajících vytvoříme pomocí cmdletu **ConvertFrom-Csv** objekt. Určíme si vlastní hlavičku (názvy vlastností jednotlivých objektů). V tomto případě to děláme proto, že zde nechceme české znaky. Výsledkem jsou objekty s kurzy jednotlivých měn:

Zeme	Mena	Mnozstvi	Kod	Kurz
----	----	-----	---	----
Austrálie	dolar	1	AUD	20,153
Brazílie	real	1	BRL	9,944
Bulharsko	lev	1	BGN	13,129

Pokud bychom si vytvořili převodní funkci, mohli bychom o něco jednodušeji volat následující příklad.

```
PS C:\> ($prevod | where Kod -eq USD).Kurz
19,749
```

Jak je vidět, použili jsme novou syntaxi volání cmdletu **Where-Object**. Nahrazuje původní: `/ Where { $_.Kod -eq 'USD' }`

Invoke-WebRequest se hodí i na složitější zpracování webových stránek. Zkusme si vyhledat nějaké informace na Bing (budu hledat informace o mém účtu na Twitteru). Dotaz má tvar:

```
PS C:\> $q =  
'http://www.bing.com/search?q=twitter+makovec&qsn&form=QBLH&filt=all&pq=twitter+makovec&sc=0-11&sp=-1&sk='
```

```
PS C:\> $a = iwr $q
```

```
PS C:\>  
$a.ParsedHtml.getElementById('results_container').GetElementsByTagName('div')  
| where ClassName -eq 'sa_cc' | Select -First 2 -Property innerText
```

```
innerText
```

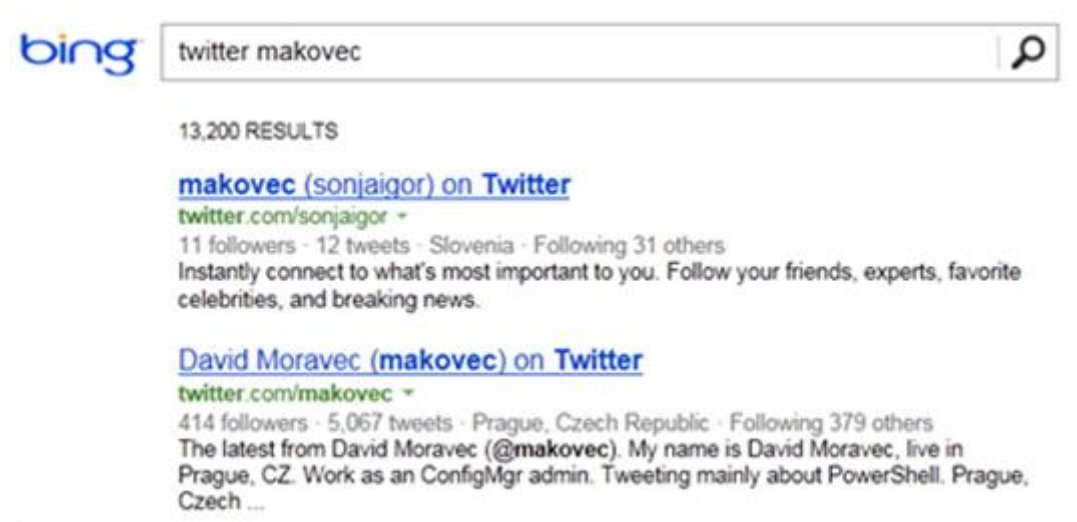
```
-----
```

```
makovec (sonjaigor) on Twitter...
```

```
David Moravec (makovec) on Twitter...
```

Malá poznámka – v tuto chvíli vím, že můj Twitter účet je na druhém místě (proto ona „magická konstanta“ *Select -First 2*). V případě parsování neznámého textu, bych musel ještě přidat kontrolu výsledků a porovnání proti známé hodnotě.

V proměnné *q* máme uložený link na Bing a do proměnné *a* si uložíme odpověď. Vzhledem k tomu, že používáme vlastnost *ParsedHtml*, máme přístup k jednotlivým prvkům stránky. Při krátké práci se zdrojem stránky jsem zjistil jména potřebných elementů a byl schopen je v jedné řádce získat. U výsledného textu mě pak zajímala vlastnost *innerText*, která obsahuje text elementu zbavených všech HTML značek. Výsledek na webu vypadá takto:



Vidíte, že s trochou práce se vám naplno otevírá svět webu. Určitě jste schopni vymyslet množství příkladů ze života, kde se popsané techniky hodí. Hodně zdaru při jejich implementaci.

PŘECHOD Z PŘÍKAZOVÉ ŘÁDKY

V jednom z předchozích dílů jsem zmiňoval, že pokud chcete s PowerShellem pracovat, je dobré jej používat co nejčastěji. Zkusíme se dnes podívat na některé časté operace, které jste možná byli zvyklí v příkazové řádce používat hodně často.

Active Directory

Pracovat v prostředí Windows a nepoužívat Active Directory je téměř nemožné. Hodně častou operací je vyhledávání počítačů nebo uživatelů. V prostředí příkazové řádky se používá příkaz **dsquery**. Tento příkaz můžeme nahradit několika cmdlety v PowerShellu.

Pokud chcete použít modul ActiveDirectory, nainstalujte si Remote Server Administration Tool (já budu instalovat verzi pro Windows 8). Tento nástroj (RSAT) je dostupný na <http://www.microsoft.com/en-us/download/details.aspx?id=28972>. Po instalaci budete mít zpřístupněno několik nových modulů, včetně toho pro Active Directory:

```
PS C:\> Get-Module -Name active* -ListAvailable
```

```
Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

ModuleType	Name	ExportedCommands
Manifest	ActiveDirectory	{Add-ADCentralAccessPolicyMember, Add-ADComputerServiceAccount, Add-A...

Jelikož se bavíme o dsget, pojďme se podívat, jaké cmdlety nám nabízí tento modul pro získávání informací:

CommandType	Name
Cmdlet	Get-ADAccountAuthorizationGroup
Cmdlet	Get-ADAccountResultantPasswordReplicationPolicy
Cmdlet	Get-ADCentralAccessPolicy
Cmdlet	Get-ADCentralAccessRule
Cmdlet	Get-ADClaimTransformPolicy
Cmdlet	Get-ADClaimType
Cmdlet	Get-ADComputer
Cmdlet	Get-ADComputerServiceAccount
Cmdlet	Get-ADDCCloningExcludedApplicationList
Cmdlet	Get-ADDefaultDomainPasswordPolicy
Cmdlet	Get-ADDomain
Cmdlet	Get-ADDomainController
Cmdlet	Get-ADDomainControllerPasswordReplicationPolicy
Cmdlet	Get-ADDomainControllerPasswordReplicationPolicyUsage
Cmdlet	Get-ADFineGrainedPasswordPolicy
Cmdlet	Get-ADFineGrainedPasswordPolicySubject
Cmdlet	Get-ADForest
Cmdlet	Get-ADGroup
Cmdlet	Get-ADGroupMember
Cmdlet	Get-ADObject
Cmdlet	Get-ADOptionalFeature
Cmdlet	Get-ADOrganizationalUnit
Cmdlet	Get-ADPrincipalGroupMembership
Cmdlet	Get-ADReplicationAttributeMetadata

Cmdlet	Get-ADReplicationConnection
Cmdlet	Get-ADReplicationFailure
Cmdlet	Get-ADReplicationPartnerMetadata
Cmdlet	Get-ADReplicationQueueOperation
Cmdlet	Get-ADReplicationSite
Cmdlet	Get-ADReplicationSiteLink
Cmdlet	Get-ADReplicationSiteLinkBridge
Cmdlet	Get-ADReplicationSubnet
Cmdlet	Get-ADReplicationUpToDateenessVectorTable
Cmdlet	Get-ADResourceProperty
Cmdlet	Get-ADResourcePropertyList
Cmdlet	Get-ADResourcePropertyValue
Cmdlet	Get-ADRootDSE
Cmdlet	Get-ADServiceAccount
Cmdlet	Get-ADTrust
Cmdlet	Get-ADUser
Cmdlet	Get-ADUserResultantPasswordPolicy

Opět bychom si měli uvědomit jedno ze základních pravidel PowerShellu – Verb-Noun. První část jména cmdletu určuje akci, druhá říká, nad kterým objektem chceme operovat. Pro dsget sáhne nejčastěji zřejmě při zjišťování informací o uživateli, skupinách a počítačích. Čemuž odpovídají tři cmdlety:

- Get-ADUser
- Get-ADGroup
- Get-AdComputer

Například úplně jednoduchý dotaz na členství počítače v konkrétní organizační jednotce:

```
PS C:\> dsquery computer domainroot -name mujnb
"CN=MUJNB,CN=Computers,DC=mydom,DC=local"
```

Můžeme získat jednoduše v PowerShellu takto:

```
PS C:\> Get-ADComputer mujnb
```

```
DistinguishedName : CN=MUJNB,CN=Computers,DC=mydom,DC=local
DNSHostName       : MUJNB.mydom.local
Enabled           : True
Name              : MUJNB
ObjectClass       : computer
ObjectGUID        : e7xxxxxx-yyyy-4da7-9398-1234568
SamAccountName    : MUJNB$
SID               : S-1-5-21-3165774027-4396
UserPrincipalName :
```

To samé bychom mohli provést pro zmiňované uživatele nebo skupiny.

Důležité je si uvědomit, že v PowerShellu získáváme objekty, se kterými můžeme dále pracovat v rouře. A i když jsme schopni předat data z dsget také do roury, přece jen je

objektové pojetí PowerShellu daleko flexibilnější (příklad práce dsget v rouře je dostupná v nápovědě).

Další skupinou často používaných příkazů jsou ty, určené pro vytváření objektů, zde zastoupené příkazem **dsadd**. V PowerShellu si můžeme zkontrolovat cmdlety New-AD*:

```
PS C:\> Get-Command -Module ActiveDirectory -Verb New | Format-Wide -Column 3
```

New-ADCentralAccessPolicy	New-ADCentralAccessRule	New-
ADClaimTransformPolicy		
New-ADClaimType	New-ADComputer	New-
ADDCCloneConfigFile		
New-ADFineGrainedPasswordPolicy	New-ADGroup	New-ADObject
New-ADOrganizationalUnit	New-ADReplicationSite	New-
ADReplicationSiteLink		
New-ADReplicationSiteLinkBridge	New-ADReplicationSubnet	New-
ADResourceProperty		
New-ADResourcePropertyList	New-ADServiceAccount	New-
ADUser		

A stejný princip bychom mohli použít pro modifikaci objektů v Active Directory. Příkaz **dsmod** můžeme nahradit cmdlety, které mají Verb = Set (seznam cmdletů si již dokážete určitě zobrazit sami).

Při zjišťování nastavení počítačů v doméně jsou častými příkazy **gpupdate** a **gpresult**. Ve Windows Serveru 2012 se objevily některé nové cmdlety a jedním z nich je **Invoke-GPUUpdate**. Náhrada za gpresult byla již v předchozí verzi serveru, jedná se o **Get-GPResultantSetOfPolicy**.

Mimochodem – modul **GroupPolicy** obsahuje nyní 28 cmdletů a velice podrobně pokrývá oblast skupinových politik. Jedná se určitě o další modul vhodný k prozkoumání:

```
PS C:\> gcm -Module GroupPolicy | fw -c 3
```

Get-GPPermissions	Set-GPPermissions	Backup-GPO
Copy-GPO	Get-GPInheritance	Get-GPO
Get-GPOReport	Get-GPPermission	Get-
GPPrefRegistryValue		
Get-GPRegistryValue	Get-GPResultantSetOfPolicy	Get-GPStarterGPO
Import-GPO	Invoke-GPUUpdate	New-GPLink
New-GPO	New-GPStarterGPO	Remove-GPLink
Remove-GPO	Remove-GPPrefRegistryValue	Remove-
GPRegistryValue		
Rename-GPO	Restore-GPO	Set-GPInheritance
Set-GPLink	Set-GPPermission	Set-
GPPrefRegistryValue		
Set-GPRegistryValue		

Ping, ipconfig, nslookup

Tyto tři příkazy patří určitě mezi nejvíce používané při práci administrátora. O pingu jsem již psal v jednom z předchozím dílů a dnes bych rád ukázal náhradu zbývajících dvou.

Pro ipconfig máme náhradu ve několika cmdletech, v závislosti na použitém přepínači. Pro přehlednost si je ukážeme v tabulce.

Ipconfig /all	Get-NetIPConfiguration –Detailed
Ipconfig /flushdns	Clear-DnsClientCache
Ipconfig /displaydns	Get-DnsClientCache
Ipconfig /registerdns	Register-DnsClient

Pro náhradu za nslookup lze použít cmdlet z modulu **DnsClient**, zkuste si tipnout který z následujících:

```
PS C:\> gcm -m DnsClient | fw -c 3
```

Add-DnsClientNrptRule	Clear-DnsClientCache	Get-
DnsClient		
Get-DnsClientCache	Get-DnsClientGlobalSetting	Get-
DnsClientNrptGlobal		
Get-DnsClientNrptPolicy	Get-DnsClientNrptRule	Get-
DnsClientServerAddress		
Register-DnsClient	Remove-DnsClientNrptRule	Set-
DnsClient		
Set-DnsClientGlobalSetting	Set-DnsClientNrptGlobal	Set-
DnsClientNrptRule		
Set-DnsClientServerAddress	Resolve-DnsName	

Pokud jste zkusili poslední z vypsanych, máte pravdu. Použití je pak opravdu jednoduché:

```
PS C:\> Resolve-DnsName www.microsoft.cz
```

Name	Type	TTL	Section	NameHost
----	----	---	-----	-----
www.microsoft.cz	CNAME	3221	Answer	websrv1.microsoft.cz

```
Name          : websrv1.microsoft.cz
QueryType     : A
TTL           : 3221
Section       : Answer
IP4Address    : 93.99.58.66
```

```
Name          : microsoft.cz
QueryType     : SOA
TTL           : 3221
Section       : Authority
NameAdministrator : msnhst.microsoft.com
```

```
SerialNumber      : 2010120201
TimeToZoneRefresh : 1800
TimeToZoneFailureRetry : 900
TimeToExpiration  : 2419200
DefaultTTL        : 3600
```

Všimněte si, že výsledkem jsou tři různé typy objektů – záznamy pro PTR, A, SOA. Pokud chcete pouze určitý záznam, můžete jej uvést v parametru **Type**.

A na závěr malý úkol do příštího dílu: Zkuste najít cmdlet, který byste použili místo **tracert**.

Pokud jej správně vyřešíte, můžete se těšit na malou odměnu

Dnes jsme se lehce dotkli oblasti, která je z mého pohledu velice důležitá – zkusit najít a používat cmdlety v situacích, kdy naše prsty automaticky sahají po starých příkazech. Opět chci připomenout starou pravdu: *Když to funguje, nesahej na to*. Ovšem v některých

případech také platí: *Pokrok nezastavíš*

POWERSHELL A AZURE ACTIVE DIRECTORY

Ještě než se pustíme do dnešního tématu, vrátil bych se rád ke konferenci TechNet NA 2013, která proběhla před pár dny v New Orleans. A to z toho důvodu, že byly odhaleny některé vlastnosti nové verze PowerShellu (pravděpodobně bude označen jako v4). Bude totiž obsahovat funkcionalitu, kterou tvůrci nazvali Desired State Configuration (DSC). DSC umožňuje vytvořit skript, který určí požadovaný stav systému a pokud systém tento stav nesplňuje, je okamžitě updatován. Syntaxe je velmi podobná syntaxi pro workflow ve verzi 3. Toto je například velmi jednoduchá definice DSC pro ověření instalace (či instalaci) webového serveru:

```
Configuration NejakýWeb
{
    Node ("Web1")
    {
        WindowsFeature IIS
        {
            Ensure = "Present"
            Name = "Web-Server"
        }
    }
}
```

Jelikož se jedná o velice zajímavou věc, doporučuji vám podívat se na záznam prezentace na Channel9: <https://channel9.msdn.com/Events/TechEd/NorthAmerica/2013/MDC-B302>.

V minulém díle jsem slíbil malý dárek. Pokud by vás zajímalo více příkladů na přechod z cmd do PowerShellu, můžete se podívat na následující soubor: <https://aka.ms/PsCmdGuideAD>

Jak se připojit do Azure Active Directory (AAD)

Již v dubnovém Flashi ukazoval mistr Skriptík cmdlet **Connect-MsolService**. Tento cmdlet je součástí modulu MSOnline (tento modul a postup jeho instalace je popsán na <https://aka.ms/aadposh>). Pro připojení do AAD zadejte zároveň přihlašovací údaje oprávněné osoby (pokud to není váš aktuálně použitý účet).

```
PS> $cred = Get-Credential admin@myFirm.onmicrosoft.com
PS> Connect-MsolService -Credential $cred
PS> Get-MsolDomain
```

Vzhledem k tomu, že **Connect-MsolService** nedá nijak vědět, že je připojen (žádná zpráva = dobrá zpráva), můžete si pomoci **Get-MsolDomain** alespoň ověřit, že vidíte vaše domény. Pokud by došlo k jakémukoli problému při připojování, zobrazí se chybové hlášení:

```
PS> Connect-MsolService -Credential $cred
```

```
Connect-MsolService : Unable to authenticate your credentials. Make sure that
your user name is in the format: <username>@<domain>. If this issue persists,
contact Support.
```

```
At line:1 char:1
```

```
+ Connect-MsolService -Credential $cred
```

```
+ ~~~~~
```

```
+ CategoryInfo          : OperationStopped: (:) [Connect-MsolService],
MicrosoftOnlineException
```

```
+ FullyQualifiedErrorId :
```

```
0x80048862,Microsoft.Online.Administration.Automation.ConnectMsolService
```

Nejvíce operací můžeme v AAD provádět s uživateli, doménami a skupinami (což samozřejmě

odpovídá funkcionalitě AAD).

```
PS> Get-Command -Module MSOnline | Group Noun | Sort Count -Desc | ft Count,
Name -Auto
```

```
Count Name
```

```
-----
```

```
5 MsolUser
5 MsolDomain
4 MsolGroup
4 MsolServicePrincipal
3 MsolFederatedDomain
3 MsolServicePrincipalCredential
3 MsolGroupMember
3 MsolRoleMember
2 MsolPartnerInformation
2 MsolPasswordPolicy
2 MsolContact
2 MsolDomainFederationSettings
```

Zbývající cmdlety se vyskytují pro daný typ už pouze jednou.

Pro práci s uživateli máme k dispozici

```
PS> Get-Command -Noun MsolUser
```

```
CommandType Name
```

```
-----
```

```
Cmdlet      Get-MsolUser
Cmdlet      New-MsolUser
Cmdlet      Remove-MsolUser
Cmdlet      Restore-MsolUser
Cmdlet      Set-MsolUser
```

Což odpovídá dostupným cmdletům z on-premise AD:

```
PS> Get-Command -Noun ADUser
```

```
CommandType Name
```

```
-----
```

```
Cmdlet      Get-ADUser
Cmdlet      New-ADUser
```

```
Cmdlet      Remove-ADUser
Cmdlet      Set-ADUser
```

A chybějící **Restore-MsolUser** je v AD zastoupen cmdletem **Restore-ADObject**. Jednou z nejčastějších operací v AAD bude nastavení zapomenutého hesla pro určitého uživatele:

```
PS> Set-MsolUserPassword -UserPrincipalName user@myFirm.onmicrosoft.com -
NewPassword HESLØ
```

Pokud zadáte změnu hesla tímto způsobem, bude uživatel po přihlášení vyzván k nastavení vlastního hesla (jako když v doméně nastavíte *User must change password at next logon*). Pokud tuto vlastnost nechcete, můžete specifikovat další parametr:

```
PS> Set-MsolUserPassword -UserPrincipalName user@myFirm.onmicrosoft.com -
NewPassword HESLØ -ForceChangePassword:$false
```

Pokud byste chtěli nastavit heslo pro všechny uživatele v doméně, můžete použít následující příkaz:

```
PS> Get-MsolUser -All | Set-MsolUserPassword -NewPassword HESLØ
```

Všimněte si, že AAD používá switch **All** pro zobrazení všech uživatelů. Jestliže chcete pro některé účty nastavit expiraci hesla na *never*, můžete použít **Set-MsolUser**.

```
PS> Set-MsolUser -UserPrincipalName user@myFirm.onmicrosoft.com -
PasswordNeverExpires $true
```

Co když chcete nejdříve zjistit, které účty mají nastaveno *PasswordNeverExpires*? Ještě stále si vystačíme se základními cmdlety:

```
PS> Get-MsolUser -All |? PasswordNeverExpires -eq $true | Select
UserPrincipalName, PasswordNeverExpires
```

Tento cmdlet má několik dalších parametrů, které opět odpovídají zvyklostem z AD, jedná se o (výpis jsem drobně zkrátil):

```
PS> (Get-Command -Name Set-MsolUser | Select -ExpandProperty Parameters).Keys
| Sort
AlternateEmailAddresses
AlternateMobilePhones
BlockCredential
City
Country
Department
DisplayName
Fax
FirstName
ImmutableId
LastName
MobilePhone
ObjectId
Office
PasswordNeverExpires
PhoneNumber
PostalCode
PreferredLanguage
SoftDeletionTimestamp
State
StreetAddress
StrongPasswordRequired
TenantId
Title
```

UsageLocation
UserPrincipalName

Pokud již máte uživatele vytvořené, můžete je začít přiřazovat do skupin. Vzhledem k tomu, že cmdlet **Add-MSolGroupMember** potřebuje ID objektu, se kterým pracuje, je dobré si nejdříve uložit skupinu a uživatele do proměnných a poté je použít. Ideálně bychom pro přiřazení více uživatelů do skupiny prováděli následující příkaz ve smyčce, např. *foreach (\$user in (Get-MSolUser -All))*:

```
PS> Add-MSolGroupMember -GroupObjectId $user.ObjectId -GroupMemberObjectId $group.ObjectId
```

Závěrem

Cílem dnešního článku bylo ukázat, že správa AAD (O365) se v ničem neliší od správy on-premise Active Directory (pokud se bavíme o PowerShellu, samozřejmě). Pokud jednou máte základy, dokážete velice jednoduše přenést vaše znalosti do jiného prostředí. A pokud se bavíme o O365, můžete vaše PowerShell-Fu použít například i při správě Exchange Online.

POWERSHELL V4

V minulém TechNet Flash magazínu jsem na začátku zmiňoval jednu z nových vlastností PowerShellu v4. Jelikož jsou od té doby již dostupné (v preview) nové verze klienta i serveru (Windows 8.1 a Windows Server 2012 R2), rád bych se na aktuální novinky podíval dnes trochu podrobněji. Pokud si budete chtít vyzkoušet popisované novinky, můžete využít odkazů publikovaných na českém TechNetu pro [Windows 8.1](#) a [Server](#) nebo si můžete rozbíhat virtuální počítač ve [Windows Azure](#), například [pomocí tohoto návodu](#).

Pokud byste chtěli nainstalovat nový PowerShell na starší typy počítačů (Windows 7 SP1, Server 2008 R2 SP1, Server 2012), můžete si stáhnout [Windows Management Framework 4.0 Preview](#).

Pojďme se podívat na některé ze zajímavých nových změn.

Save-Help - tento cmdlet se dočkal změny, po které volalo mnoho lidí. Pokud jste měli počítač, na kterém nebyl nainstalován nějaký PowerShell modul, nemohli jste pro tento modul nainstalovat nápovědu. Tato funkcionality se hodí v případě, že jste chtěli mít na svém počítači nápovědu pro serverové moduly (pro studijní důvody) nebo jste potřebovali stáhnout nápovědu na server, který neměl konektivitu do internetu.

Příklad si ukážeme na updatu helpu v mém labu. Mám zde Active Directory doménu, kde můj DC počítač nemá přístup na internet. Na DC spustím tento příkaz:

```
PS> Export-Clxml -Path c:\Temp\dns.xml -InputObject (Get-Module -Name DnsServer -List)
```

Ve výsledném XML souboru je vyexportována struktura modulu pro správu DNS. Na následujícím obrázku je zobrazen pouze začátek souboru:

```
<?xml version="1.0"?>
<Obj xmlns="http://schemas.microsoft.com/powershell/2004/04" Version="1.1.0.1">
  - <Obj RefId="0">
    - <TN RefId="0">
      <T>ModuleInfoGrouping</T>
      <T>System.Management.Automation.PSModuleInfo</T>
      <T>System.Object</T>
    </TN>
    <ToString>DnsServer</ToString>
    - <Props>
      <B N="LogPipelineExecutionDetails">false</B>
      <S N="Name">DnsServer</S>
      <S N="Path">C:\Windows\system32\WindowsPowerShell\v1.0\Modules\DnsServer\DnsServer.psd1</S>
      <S N="Definition"/>
      <S N="Description"/>
      <G N="Guid">46f598e5-9907-42b2-afbb-68e5f7e34604</G>
      <S N="HelpInfoUri">http://go.microsoft.com/fwlink/?LinkID=217875</S>
      <S N="ModuleBase">C:\Windows\system32\WindowsPowerShell\v1.0\Modules\DnsServer</S>
      <Nil N="PrivateData"/>
      <Version N="Version">1.0.0.0</Version>
    - <Obj RefId="1" N="ModuleType">
      - <TN RefId="1">
        <T>System.Management.Automation.ModuleType</T>
        <T>System.Enum</T>
        <T>System.ValueType</T>
        <T>System.Object</T>
      </TN>
      <ToString>Manifest</ToString>
      <I32>2</I32>
    </Obj>
    <S N="Author">Microsoft Corporation</S>
  </Obj>
</Obj>
```

Tento soubor si přeneseme na počítač, který má připojení do internetu a spustíme příkaz pro uložení nápovědy:

```
PS> $xml = Import-Clixml c:\Scripts\dns.xml
PS> Save-Help -Module $xml -Destination c:\Scripts\DnsHelp
```

Uložené soubory přeneseme zpět na zdrojový počítač a zde je již naimportujeme pomocí **Update-Help**

```
PS> Update-Help -Module DnsServer -Source c:\Temp\DnsHelp
```

Toto zřejmě není operace, kterou byste prováděli příliš často (někteří možná nápovědu takto aktualizovat nepotřebují), ale pro určité scénáře se jedná o zajímavou cestu.

Malé změny v cmdletech

Pro práci s naplánovanými úlohami určitě používáte cmdlety **Register-ScheduledJob** a **Set-ScheduledJob**. Tyto dva cmdlety nyní obsahují nový parametr – **RunNow**. Pomocí toho parametru můžete úlohu spustit okamžitě a není potřeba vytvářet trigger.

```
-RunNow [<SwitchParameter>]
```

Starts a job immediately, as soon as the Register-ScheduledJob cmdlet

is run. This parameter eliminates the need to trigger Task Scheduler to

run a Windows PowerShell script immediately after registration, and does

not require users to create a trigger that specifies a starting date and time.

Required? false

Position? named
Default value false
Accept pipeline input? false
Accept wildcard characters? False

Když už jsme u těch triggerů J Pro cmdlety **New-JobTrigger** a **Set-JobTrigger** přibyl také nový parametr: **RepeatIndefinitely**.

-RepeatIndefinitely [<SwitchParameter>]

This parameter, available starting in Windows PowerShell 4.0, eliminates the necessity of specifying a TimeSpan.MaxValue value for the repetitionDuration parameter to run a scheduled job repeatedly, for an indefinite period.

Po instalaci PowerShellu na starších verzích jste museli pro běh skriptů nastavit Execution Policy. Ve Windows Serveru 2012 R2 je tato politika nastavena na **RemoteSigned** (netýká se Windows 8.1). Je zde vidět posun v administraci serverů směrem k jednodušší administraci. I když Microsoft nemá tuto změnu označenou jako „breaking“, dle mého názoru je potřeba, aby o této změně byli lidé informováni – i když se primárně nejedná o žádné narušení bezpečnosti.

Nového parametru se dočkal i můj oblíbený cmdlet **Get-Process**. Jedná se o parametr **IncludeUserName** a jak jméno naznačuje, do výpisu přidá jméno uživatele, pod kterým byl proces spuštěn:

```
PS> Get-Process | select -First 10
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
57	7	1940	8772	54	1.75	668	conhost
135	10	1616	3680	42	0.38	388	csrss
88	9	1128	3512	39	0.02	440	csrss
112	10	1656	30488	65	0.69	2664	csrss
201	15	22812	32644	95	0.11	728	dwm
192	18	13320	54212	139	1.91	2768	dwm
881	65	17004	55588	342	2.30	1968	explorer
0	0	0	24	0		0	Idle
270	21	8324	23332	128	0.14	1384	LogonUI
782	18	3320	9504	35	0.50	544	lsass

```
PS> Get-Process -IncludeUserName | select -First 10
```

Handles	WS(K)	VM(M)	CPU(s)	Id	UserName	ProcessName
57	8772	54	1.69	668	MAKOVECSVRR2-01\makovec	conhost
135	3680	42	0.38	388		csrss
88	3512	39	0.02	440		csrss
112	30488	65	0.69	2664		csrss

201	32644	95	0.11	728	Window Manager\DWM-1	dwm
192	54212	139	1.91	2768	Window Manager\DWM-2	dwm
881	55588	342	2.30	1968	MAKOVECSVRR2-01\makovec	explorer
0	24	0		0		Idle
270	23332	128	0.14	1384	NT AUTHORITY\SYSTEM	LogonUI
782	9492	34	0.50	544	NT AUTHORITY\SYSTEM	lsass

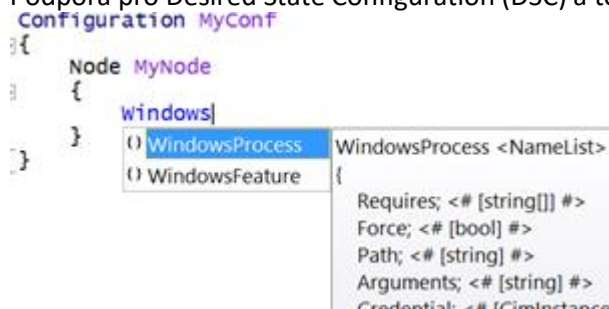
A také jsme se dočkali nového cmdletu, jedná se o **Get-FileHash**. Hash můžete počítat jedním z následujících algoritmů: SHA1, SHA256, SHA384, SHA512, MACTripleDES, MD5, RIPEMD160 (defaultní je SHA256). Pokud chcete mít standardní jiné, můžete použít mou (další oblíbenou „fíčuru“ – PSDefaultParameterValues, např. takto:

```
PS> $PSDefaultParameterValues = @{'Get-FileHash:Algorithm'='MD5'}
PS> Get-FileHash c:\Temp\dns.xml | fl *
```

```
Path : C:\Temp\dns.xml
Type : System.Security.Cryptography.MD5CryptoServiceProvider
Hash : FUNAxqeXLOkiR0yIGcF1ag==
```

Vzhledem k některým rozšířením jazyka, došlo i ke změnám v PowerShell ISE. Nejedná se ovšem o žádné převratné novinky. Jsou to spíše maličkosti, které zjednoduší práci, např:

- Podpora pro debugging workflow
- Podpora pro Desired State Configuration (DSC) a to v oblasti IntelliSense



- Podpora pro debugging vzdálených skriptů

Zároveň byla odstraněna chyba, která byla velmi otravná, a narazilo na ní množství lidí. V případě, že jste v ISE spustili cmdlet **Invoke-WebRequest** a po vrácení hodnot ISE zavřeli, proces běžel stále na pozadí a po nějaké době vám byl schopen obsadit celou paměť. Několikrát se mi stalo, že jsem si říkal, proč mám tak pomalý počítač a až pohled na Process Monitor odhalil smutnou pravdu. Například teď při testování mám ISE dávno vypnuté, ale pořád mi běží v procesech a celkem kvalitně vytěžuje procesor:

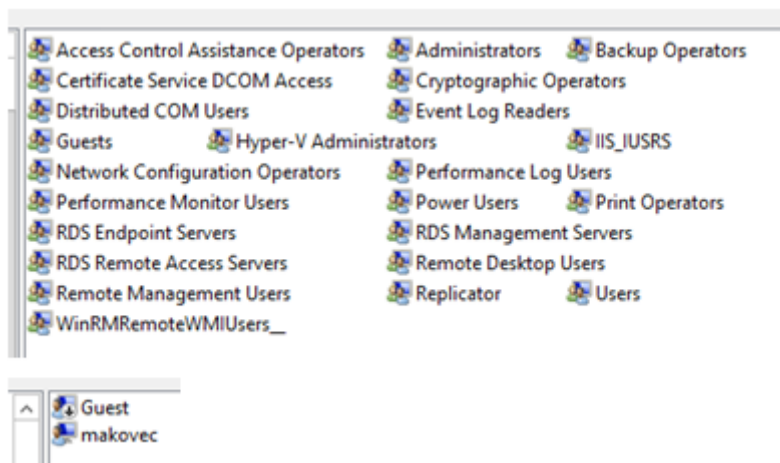
Process	CPU
powershell_ise.exe	24.74

Naštěstí, tohle už nás ve verzi 4 nečeká.

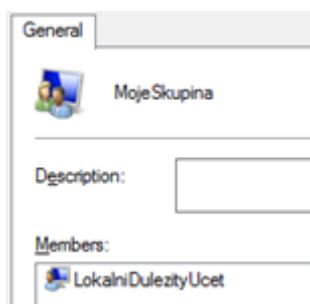
Desired State Configuration (DSC)

Jak jsem již naznačoval minule, pro mne největší novinkou je DSC. Zde si můžete určit daný stav systému a ten poté pomocí DSC vynucovat. Dnes si ukážeme krátkou ukázkou a podrobnější popis vyjde zřejmě na příští pokračování.

Naším cílem bude zajistit, že na počítači bude vždy lokální účet daného jména (LokalniDulezityUcet) a tento účet bude vždy členem určité skupiny (MojeSkupina). Na začátku operace neexistuje ani účet ani skupina:



Po spuštění DSC se objeví jak účet, tak i skupina, které je nyní členem:



Celé kouzlo spočívá v následující konfiguraci:

```
Configuration MyUserConfig
{
    param($MachineName)

    Node $MachineName
    {
        User MyUser
        {
            Ensure = 'Present'
            UserName = 'LokalniDulezityUcet'
        }
    }
}
```

```

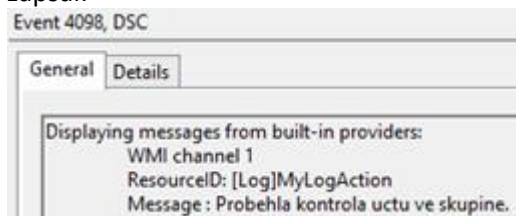
Group MyGroup
{
    Ensure = 'Present'
    Name = 'MojeSkupina'
    MembersToInclude = 'LokalniDulezityUcet'
    Requires = '[User]MyUser'
}

Log MyLogAction
{
    Message = 'Probehla kontrola uctu ve skupine'
    Requires = '[Group]MyGroup'
}
}
}

```

Zde jsme si v rámci **Configuration** vytvořili **Node**, který obsahuje chtěnou definici cílového stavu.

- **Uživatel** je definován jako chtěný „Present“ a uvádíme jeho jméno.
- **Skupina** je také potřebná, její jméno je *MojeSkupina* a chceme přidat uživatele vytvořeného v předchozím kroku. Abychom zajistili, že uživatel opravdu existuje, použili jsme *Requires* a uvedli sekci, která má být spuštěna před aktuální.
- **Logování** probíhá do Analytic logu a text, který jsme chtěli mít uvedený se tam opravdu zapsal:



Po vytvoření konfigurace je potřeba zkompilevat tuto konfiguraci do MOF souboru a poté ji spustit. Vše provedeme následujícími dvěma příkazy:

```

PS> MyUserConfig -MachineName $env:COMPUTERNAME
PS> Start-DscConfiguration -Path .\MyUserConfig

```

Pokud bych nyní smazal uživatelský účet a spustil DSC znovu, účet se vytvoří a přidá do skupiny. Což si můžeme ověřit, pokud spustíme kontrolu s parametry **Wait** a **Verbose**.

```

PS> Start-DscConfiguration -Path .\MyUserConfig -Wait -Verbose
VERBOSE: Perform operation 'Invoke CimMethod' with following
parameters, 'methodName' = SendConfigurationApply,'className' =
MSFT_DSCLocalConfigurationMan
ager,'namespaceName' =
root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer MAKOVECSVRR2-01

```

```

with user sid S-1-5-21-1791083104-4223022288-313497419-500.
VERBOSE: 'DSCEngine': Starting to process the Set request.
VERBOSE: 'DSCEngine': Starting to process resource. '[User]MyUser'
VERBOSE: 'DSCEngine': Performing the test operation. '[User]MyUser'
VERBOSE: 'DSCEngine': [User]MyUser: The Test operation took 0.3590
seconds.
VERBOSE: 'DSCEngine': Performing Set operation. '[User]MyUser'
VERBOSE: '[User]MyUser': Configuration of user LokalniDulezityUcet
started.
VERBOSE: '[User]MyUser': User LokalniDulezityUcet created
successfully.
VERBOSE: '[User]MyUser': Configuration of user LokalniDulezityUcet
completed successfully.
VERBOSE: 'DSCEngine': [User]MyUser: The Set operation took 0.4690
seconds.
VERBOSE: 'DSCEngine': The resource finished processing.
'[User]MyUser'
VERBOSE: 'DSCEngine': Starting to process resource. '[Group]MyGroup'
VERBOSE: 'DSCEngine': Performing the test operation.
'[Group]MyGroup'
VERBOSE: 'DSCEngine': [Group]MyGroup: The Test operation took 0.4060
seconds.
VERBOSE: 'DSCEngine': Performing Set operation. '[Group]MyGroup'
VERBOSE: '[Group]MyGroup': Group MojeSkupina properties updated
successfully.
VERBOSE: 'DSCEngine': [Group]MyGroup: The Set operation took 0.4380
seconds.
VERBOSE: 'DSCEngine': The resource finished processing.
'[Group]MyGroup'
VERBOSE: 'DSCEngine': Starting to process resource.
'[Log]MyLogAction'
VERBOSE: 'DSCEngine': Performing the test operation.
'[Log]MyLogAction'
VERBOSE: 'DSCEngine': [Log]MyLogAction: The Test operation took
0.0150 seconds.
VERBOSE: 'DSCEngine': Performing Set operation. '[Log]MyLogAction'
VERBOSE: '[Log]MyLogAction': Probehla kontrola uctu ve skupine
VERBOSE: 'DSCEngine': [Log]MyLogAction: The Set operation took
0.0000 seconds.
VERBOSE: 'DSCEngine': The resource finished processing.
'[Log]MyLogAction'
VERBOSE: 'DSCEngine': Set request completed.
VERBOSE: DSCEngine: The total operation took 1.8900 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 2.115
seconds

```

Tím bych pro dnešek PowerShell opustil. Těším se na vaše komentáře o DSC

HRÁTKY S HYPER-V

Dnes bych se rád zaměřil trochu více na zkoumání fungování PowerShellu. Čím dál tím častěji se mi stává, že některý z klientů či kolegů potřebuje pomoci s kouskem kódu v PowerShellu. Většinou pro správu mě „neznámé“ technologie. Neznámé = není mou primární oblastí zájmu (SharePoint, Lync, Exchange, ...). Pak je pro mne nutné co nejdříve proniknout do fungování produktu a zjistit, jak vlastně mohu PowerShell v určitém scénáři použít. Nedávno jsem si vzpomněl na jednu mou starou příhodu s Hyper-V. Zkusím na příkladu demonstrovat, jak jsem postupoval a budu rád, když kolem tématu vznikne diskuse.

Předesílám, že jsem si zadal, že úkol vyřeším pouze pomocí PowerShellu a nebudu otevírat prohlížeč. I když – možná jsem mohl trochu podvádět a použít:

```
PS> Invoke-WebRequest -Uri 'http://www.bing.com/search?q=hyper-v+snapshot' |  
Select -Expand links | where href -like 'http*' | Format-List innerText, href  
  
innerText : Hyper-V Virtual Machine Snapshots: FAQ - Resources and Tools for  
...  
href      : http://technet.microsoft.com/en-us/library/dd560637(v=WS.10).aspx  
.. zkráceno
```

A výsledky dále zpracovat ve smyčce. Možná si to nechám jako zajímavé cvičení pro příště.

Potřeboval jsem si uložit VHD soubory pro všechny své virtuální počítače a tyto soubory pak poslat dál pro další použití. Rozhodl jsem se udělat ze všech VM snapshoty a aktuální verzi poté vyexportovat. Vezměme si tedy Hyper-V modul jako černou skříňku, o které toho moc nevíme. Úmyslně budu v následujících řádcích používat aliasy a další „zkracovadla“ tak, jak jsem je (asi – přeci jen je to již dlouho) tenkrát použil.

```
PS>gmo hyper-v
```

ModuleType	Name	ExportedCommands
Binary	Hyper-V	{Add-VMdvdDrive, Add-VMFibreChannelHba, ...

Aha – modul bychom měli, co s ním vlastně můžeme dělat?

```
PS>gcm -m hyper-v
```

CommandType	Name	ModuleName
Cmdlet	Add-VMdvdDrive	Hyper-V
Cmdlet	Add-VMFibreChannelHba	Hyper-V
Cmdlet	Add-VMHardDiskDrive	Hyper-V

... dlouhý seznam

Zajímavé, kolik tam toho je vlastně schováno?

```
PS>gcm -m hyper-v | measure
```

Count : 164

Poznámka: V následujících výpisech budu listingy hodně zkracovat a již na to nebudu dále upozorňovat. Zkuste si projít stejný postup např. pro jiný modul.

Hmm, co všechno můžu dělat, aniž bych cokoli pokazil?

```
PS>gcm -verb get -m hyper-v | fw -c 4
```

```
Get-VHD Get-VM Get-VMbios Get-VMComPort Get-VMConnectAccess Get-VMdvdDrive  
Get-VMFibreChannelHba Get-VMFloppyDiskDrive
```

Vypadá to na slušný seznam, který obsahuje 40 (ověřeno dalším příkazem) cmdletů. Takže jaké tu mám VM?

```
PS>get-vm
```

Name	State	CPUUsage(%)	MemoryAssigned(M)
Uptime	Status		
Emulator WVGA 512MB.dmoravec	Off	0	0
Operating normally			
MT-TST-Win2012	Off	0	0
Operating normally			
Windows Server 2012 R2	Off	0	0
Operating normally			

Jak můžu dál použít tento cmdlet (zde můžu projít tabulátorem dostupné parametry nebo spustit help).

```
PS>help get-vm
```

SYNTAX

```
Get-VM [[-Name] <String[]>] [-ComputerName <String[]>]  
[<CommonParameters>]
```

```
Get-VM [-ClusterObject] <PSObject> [<CommonParameters>]
```

```
Get-VM [[-Id] <Guid>] [-ComputerName <String[]>] [<CommonParameters>]
```

OK – **Name** vypadá jako zajímavý parametr. Jak jsem již několikrát upozorňoval – ve většině případů vám PowerShell ve standardním zobrazení nevrátí všechny vlastnosti objektu, proto je dobrou praxí používat např. u cmdletu **Format-List** parametr **Force**.

```
PS>get-vm -Name mt-tst-win2012 | fl *
```

```
VMName           : MT-TST-Win2012  
VMIId            : 8fe1a348-310b-4e28-b6cd-3e015809cbc0  
Id               : 8fe1a348-310b-4e28-b6cd-3e015809cbc0  
Name             : MT-TST-Win2012  
State            : Off  
OperationalStatus : {Ok}  
PrimaryOperationalStatus : Ok  
SecondaryOperationalStatus :  
StatusDescriptions : {Operating normally}  
ParentSnapshotId  :  
ParentSnapshotName :
```

Výpis nám vrátí téměř 60 různých vlastností. Z výpisu je vidět, že tato VM nemá zatím žádný snapshot. Dobrá, můžeme tedy zkusit nějaký vytvořit.

Z předchozích výpisů víme, že existuje cmdlet **Get-VMSnapshot**, tak si zkusíme zobrazit všechny cmdlety pracující se snapshoty.

```
PS>gcm -m hyper-v -No vmsnapshot
```

CommandType	Name	ModuleName
Cmdlet	Export-VMSnapshot	Hyper-V
Cmdlet	Get-VMSnapshot	Hyper-V
Cmdlet	Remove-VMSnapshot	Hyper-V
Cmdlet	Rename-VMSnapshot	Hyper-V
Cmdlet	Restore-VMSnapshot	Hyper-V

Hmm, ve výpisu evidentně chybí očekávaný cmdlet **New-VMSnapshot**. Není zde ani žádný Set ani jiný podobný. Co nám říká help u **Get-VMSnapshot**? Ve většině nápověd obsahuje sekce RELATED LINKS cmdlety, které jsou v nějakém vztahu k zobrazovanému.

```
PS>help get-vm
```

RELATED LINKS

REMARKS

Tak zde by zasloužil Hyper-V team poprvé za uši. Pohledem do ostatních cmdletů zjistíme, že ani v nich nic není. Tak zkusíme obecnější dotaz do nápovědy:

```
PS>help snapshot
```

Name	Category	Module
Export-VMSnapshot	Cmdlet	Hyper-V
Get-VMSnapshot	Cmdlet	Hyper-V

Hmm, také nic. Že by to nešlo? To mi přijde divné. V tomto kroku bych v normálním procesu otevíral prohlížeč a potřebný cmdlet bych našel během pár vteřin. Ale předsevzetí je předsevzetí. V jednom z předchozích výpisů jsme našli vlastnost popisující snapshot. Co se tak podívat na cmdlety dostupné pro vlastní VM?

```
PS>gcm -m hyper-v -no vm
```

CommandType	Name	ModuleName
Cmdlet	Compare-VM	Hyper-V
Cmdlet	Export-VM	Hyper-V
Cmdlet	Get-VM	Hyper-V
Cmdlet	Checkpoint-VM	Hyper-V
Cmdlet	Import-VM	Hyper-V

Ah – jeden z cmdletů má trošku povědomé jméno:

```
PS>help checkpoint-vm
```

NAME

Checkpoint-VM

SYNOPSIS

Creates a snapshot of a virtual machine.

Voila a uff. Skvělé, co tedy dál:

```
PS>help checkpoint-vm -exa
```

Example 2

```
PS C:\>Get-VM Test -ComputerName Server1 | Checkpoint-VM
```

Checkpoints virtual machine Test on Hyper-V host Server1.

Toto přesně potřebujeme. Pokus číslo 1. A další několikrát zmiňovaná pravda: parametr **WhatIf** je náš nejlepší přítel:

```
PS>get-vm | Checkpoint-VM -SnapshotName ToExport -WhatIf
```

What if: Checkpoint-VM will create a snapshot for virtual machine "Emulator WVGA 512MB.dmoravec".

What if: Checkpoint-VM will create a snapshot for virtual machine "MT-TST-Win2012".

What if: Checkpoint-VM will create a snapshot for virtual machine "Windows Server 2012 R2".

No, je čas na trochu dobrodružství:

```
PS>get-vm | Checkpoint-VM -SnapshotName ToExport
```

```
PS>get-vm | get-vmsnapshot
```

VMName	Name	SnapshotType
CreationTime	ParentSnapshotName	
-----	----	-----
-----		-----
Windows Server 2012 R2	ToExport	Standard
1. 8. 2013 16:47:03		

Nyní je potřeba vytvořené snapshoty exportovat. Rychlým pohledem do helpu je nalezení potřebného cmdletu hračka: **Export-VMSnapshot**. Poslední kontrola:

```
PS>get-vm | Get-VMSnapshot -Name ToExport -ea 0 | Export-VMSnapshot -Path  
c:\temp -WhatIf  
What if: Export-VMSnapshot will export the snapshot "ToExport".
```

A můžeme provést finální export:

```
PS>get-vm | Get-VMSnapshot -Name ToExport -ea 0 | Export-VMSnapshot -Path  
c:\temp -Verbose  
VERBOSE: Export-VMSnapshot will export the snapshot "ToExport".
```

Pak už jen stačí vzít finální adresář a nahrát jej na potřebné místo.

Jak jsem předesílal na začátku. Dnešním cílem nebylo vás seznámit s nějakou novinkou, ale spíše s možností, jak lze s minimem informací dokončit práci, **pokud znáte základní příkazy PowerShellu**. Cmdlety **Get-***, **help**, **WhatIf** jsou vaši přátelé a můžete pomocí nich zvládnout správu libovolných technologií. Ale ještě jednou opakuji to, co je mottem dnešního článku: **Musíte zvládnout základy!** I proto je dobré občas zabrousit do starších článků a podívat se, jak například funguje systém nápovědy.

TIP NA PRÁCI S ČASEM

Dnešní článek bude takové malé zamyšlení nad plynoucím časem. Poslední dobou hodně pracuji s ConfigMgr 2012 SP1 (a už se netrpělivě těším na nový ConfigMgr 2012 R2). Jakožto člověk pracující převážně s PowerShellem se mi líbí možnost použít ConfigMgr cmdlety z konzole a „nezdržovat“ se s přechodem do GUI. I v časech ConfigMgr 2007 jsem hodně věcí řešil vlastními skripty (což mi teď připomnělo můj starý [PowerGUI PowerPack pro SMS 2003](#)).

Častou operací při práci s ConfigMgr 2012 (dále budu psát jen CM) je kontrola chyb různých komponent. Pro tento účel existuje cmdlet [Get-CMSiteStatusMessage](#). Tento cmdlet má jeden povinný parametr: **ViewingPeriod**.

```
PS MT0:\> Get-CMSiteStatusMessage -ViewingPeriod "2013/08/21 08:00:00" -  
Severity Error | Measure-Object
```

```
Count : 37  
Average :  
Sum :  
Maximum :  
Minimum :  
Property :
```

Ve výpisu jsem dostal seznam chyb, které se momentálně vyskytují v mém labu. Čas je zadaný ve formě textu (formát US). Osobně nemám zadávání času tímto způsobem moc rád, radši používám cmdlet **Get-Date** a z něj odvozené časové konstanty. Zároveň chci při práci s konzolí zadávat čas ve

tvaru „ukaz mi, co se stalo za poslední dvě hodiny“ než jej zadávat jako hodnotu. Pojdme si předchozí příklad přepsat.

```
PS MS0:\> $time = (Get-Date).AddHours(-2)
PS MT0:\> Get-CMSiteStatusMessage -ViewingPeriod $time -Severity Error |
Measure-Object | Select Count
```

Count

10

Nyní je vidět, že jsem si vytvořil proměnnou **time**, která obsahuje hodnotu času před dvěma hodinami. Tuto proměnnou poté použiji v daném cmdletu.

Touto cestou si můžeme vytvořit několik dalších časových konstant dle libosti. Můžeme se podívat, jaké konstanty v současnosti používám na jednom ze svých CM serverů:

```
PS MT0:\> dir Variable: |? value -is [datetime]
```

Name	Value
-----	-----
LastDay	6. 09. 2013 13:40:21
LastHour	7. 09. 2013 12:40:21
NextHour	7. 09. 2013 14:40:21
PredHodinou	7. 09. 2013 12:40:21
Tomorrow	8. 09. 2013 13:40:21
Vcera	6. 09. 2013 13:40:21
ZaHodinu	7. 09. 2013 14:40:21
Zitra	8. 09. 2013 13:40:21

```
PS MS0:\> $Vcera
```

6. října 2013 13:40:26

Tento kód vylistuje všechny proměnné a vypíše pouze ty, které mají hodnotu typu **DateTime**. Proto jsem mohl použít i například tuto konstrukci:

```
PS MT0:\> Get-CMSiteStatusMessage -ViewingPeriod $PredHodinou -Severity Error
| Measure-Object | Select Count
```

Count

4

Vytváření všech těchto proměnných probíhá ve funkci **prompt** a proto je čas v nich uložený „vždy“ aktuální, resp. Jedná se o čas, kdy jsem provedl předchozí příkaz. Pokud chcete opravdu naprosto přesný čas, řešení naleznete na konci článku.

V definici funkce **prompt**, mám uvedeno (uvádím pouze malou část kódu):

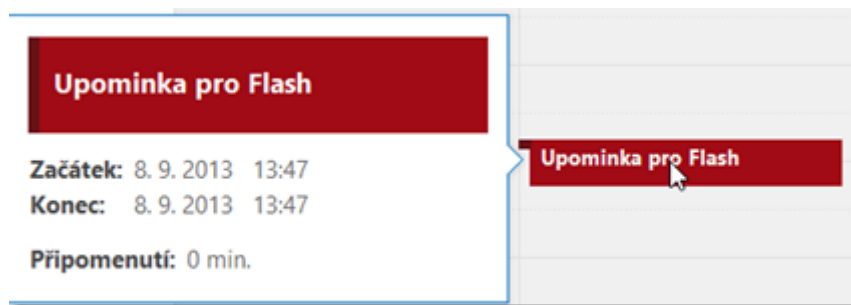
```
function prompt
{
    $Global>LastHour = [System.DateTime]::Now.AddHours(-1)
    $Global>LastDay = [System.DateTime]::Now.AddDays(-1)
    $Global:Today = [System.DateTime]::Today
    "$pwd>"
}
```

Všimněte si proměnné **Today**. Oproti standardnímu **Get-Date** obsahuje čas o půlnoci dnešního dne a proto opravdu vypíše případné chyby vzniklé po půlnoci.

Za pomoci těchto proměnných si také mohu zadávat upomínky do kalendáře. Tím, že většinu času trávím v konzoli PowerShellu (nebo ji mám opravdu nadosah), mohu použít následující konstrukci:

```
PS MT0:\> New-OutlookReminder -Subject 'Upominka pro Flash' -DateTime $Zitra
```

A opravdu se mi v kalendáři objeví upomínka na zítra:



Samozřejmě by zapisování tak dlouhého textu nebylo příjemné a proto jsem jej minimalizoval použitím aliasu a nastavením vlastností parametrů. Proto mi zápisy

```
PS MT0:\> rem 'Dalsi upominka'
```

```
PS MT0:\> rem 'A jeste jedna' (hod 3)
```

Přidají upomínku na čas za hodinu (defaultní hodnota) nebo za tři hodiny (přičemž **hod** je alias pro další funkci pracující s časovými konstantami).

Jak jsem již řekl. Hodnota uložená v proměnných je časovým údajem uloženým při posledním běhu funkce prompt. Proto může dojít k určitým odchylkám. Před několika měsíci byl na webu PowerShell.com uveden návod na vytvoření opravdové proměnné měnící svoji hodnotu za běhu:

```
$Global:Now = Set-PSBreakpoint -Variable Now -Mode Read -Action { Set-Variable Now (Get-Date) -Option ReadOnly, AllScope -Scope Global -Force }
```

Zde se využívá možnosti breakpointu nastaveného na čtení proměnné. Při tomto čtení se zároveň hodnota proměnné změní.

Doufám, že jste dnes dostali užitečný tip, jak si lze ulehčit práci v PowerShellu za pomoci jednoduchých proměnných.

TRANSAKCE

Stejně jako Mistr Skriptík, i já jsem nedávno dostal dotaz na transakce v PowerShellu. Že prý kdysi v PowerShellu byly, ale teď tam nejsou a proč byly odebrány.

Transakce v PowerShellu pořád jsou, jenom se o nich nemluví, protože jejich implementace není taková, jakou bychom si ji všichni představovali. Zatím je dostupná pouze pro registr. Nicméně pojdme se podívat, jak taková implementace vypadá.

V první řadě si řekneme, co si pod pojmem transakce představit. Jedná se o operaci, při které nám systém zajistí několik základních vlastností: Izolace (prováděné příkazy neovlivní systém – dokud to nepovolíme), Atomicita (když se rozhodneme změny uložit, uloží se buď všechny, nebo žádná), Konzistence (pokud se v průběhu operace objeví chyby, které mohou vést ke stavu, který nechceme, můžeme operaci zrušit a vrátit provedené kroky), Odolnost (pokud je operace dokončená, provedené změny jsou trvalé). I když zní předchozí pokus o popis transakčního zpracování příliš „vědecky“, žádná věda to není. Pojdme si vše ukázat na příkladu.

Nejprve zjistíme, kde můžeme transakce provádět. Jistě víte, že v PowerShellu existuje systém tzv. Providerů. Jedná se například o FileSystem Provider pro práci se souborovým systémem, dále máme např. Registry Provider, atd. Jejich seznam si můžeme zobrazit pomocí cmdletu Get-PSProvider.

```
PS C:\> Get-PSProvider
```

Name	Capabilities	Drives
----	-----	-----

Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C,
Dropbox, Download, E}		
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM,
HKCU, HKU, HKCR...}		
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{Cert}
ActiveDirectory	Include, Exclude, Filter, ShouldProcess, Crede...	{}

Pokud se chceme podívat jenom na providery podporující transakce, můžeme si výstup filtrovat:

```
PS C:\> Get-PSProvider |? Capabilities -m 'Transactions' | ft -auto
```

Name	Capabilities	Drives
Registry	ShouldProcess, Transactions	{HKLM, HKCU, HKU, HKCR...}

Opravdu, jediným providerem je zatím ten pro registr.

Ukážeme si nějakou operaci a zkusíme ji provést transakčně. Nejprve se přepneme do registru:

```
PS C:\> cd HKCU:\Software\Makovec\Tran
```

A poté odstartujeme transakci pomocí cmdletu Start-Transaction.

```
PS C:\> Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.

PowerShell nám rovnou oznamuje, že všechny operace, které chceme provést jako součást transakce, musíme volat i s parametrem **UseTransaction**. Můžeme si zobrazit všechny cmdlety, které transakce podporují:

```
PS C:\> gcm -ParameterName UseTransaction | fw -c 3
```

mkdir	Add-Content	Clear-Content
Clear-Item	Clear-ItemProperty	Convert-Path
Copy-Item	Copy-ItemProperty	Get-Acl
Get-Content	Get-ChildItem	Get-Item
Get-ItemProperty	Get-Location	Get-PSDrive
Invoke-Item	Join-Path	Move-Item
Move-ItemProperty	New-Item	New-ItemProperty
New-PSDrive	Pop-Location	Push-Location
Remove-Item	Remove-ItemProperty	Remove-PSDrive
Rename-Item	Rename-ItemProperty	Resolve-Path
Set-Acl	Set-Content	Set-Item
Set-ItemProperty	Set-Location	Split-Path
Test-Path	Use-Transaction	

A nyní si vytvoříme strukturu, do které budeme zapisovat data:

```
PS C:\> mkdir MyKey -UseTransaction
```

```
Hive: HKEY_CURRENT_USER\Software\Makovec\Tran
```

Name	Property
MyKey	

Pro kontrolu si můžeme zobrazit výpis aktuálního stavu. Transakce by ještě neměla být zapsána.

```
PS C:\> ls
```

```
PS C:\> New-Item MyKey\Datum -UseTransaction
```

```
Hive: HKEY_CURRENT_USER\Software\Makovec\Tran\MyKey
```

```
Name    Property
```

```
-----
```

```
Datum
```

A celou transakci uložíme.

```
PS C:\> Complete-Transaction
```

```
PS C:\> ls
```

```
Hive: HKEY_CURRENT_USER\Software\Makovec\Tran
```

```
Name    Property
```

```
-----
```

```
MyKey
```

Nyní už máme vše zapsáno tak, jak jsme si představovali. Ještě zapíšeme aktuální datum a čas do nově vytvořené struktury.

```
PS C:\> Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.

```
PS C:\> New-ItemProperty -Path .\MyKey\Datum -Name Datum -Value $(date)
```

```
Datum          : 5. 11. 2013 7:48:38
```

```
PSPath          :
```

```
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\MyKey\Datum
```

```
PSParentPath    :
```

```
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\MyKey
```

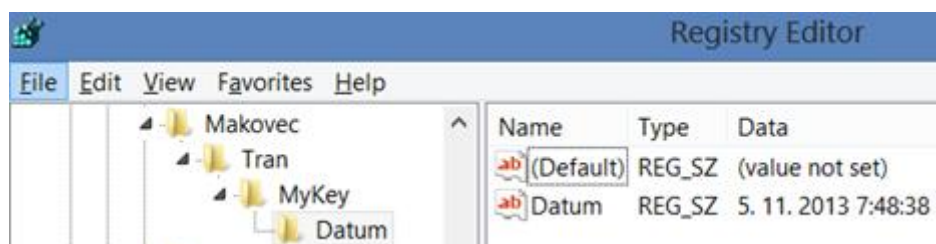
```
PSChildName     : Datum
```

```
PSDrive         : HKCU
```

```
PSProvider      : Microsoft.PowerShell.Core\Registry
```

```
PS C:\> Complete-Transaction
```

A výsledek si můžeme zobrazit pomocí regeditu:



Transakce mají výhodu v tom, že pokud dojde k chybě, výsledná operace se neprovede. Pojdme si to demonstrovat na příkladu.

```
PS C:\> Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.

```
PS C:\> New-ItemProperty -Path . -Name Datum2 -Value (get-date) -UseTransaction
```

```

Datum2 : 5. 11. 2013 8:01:48
PSPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\My
Key\Datum
PSParentPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\My
Key
PSChildName : Datum
PSDrive : HKCU
PSProvider : Microsoft.PowerShell.Core\Registry

PS C:\> New-ItemProperty -Path .\asdfg -Name Datum3 -Value (get-date) -
UseTransaction
New-ItemProperty : Cannot find path
'HKCU:\Software\Makovec\Tran\MyKey\Datum\asdfg' because it does not exist.
At line:1 char:1
+ New-ItemProperty -Path .\asdfg -Name Datum3 -Value (get-date) -
UseTransaction
+
~~~~~
+ CategoryInfo          : ObjectNotFound: (HKCU:\Software\...Key\Datum\asdfg:String)
[New-ItemProperty], ItemNotFo
undException
+ FullyQualifiedErrorId :
PathNotFound,Microsoft.PowerShell.Commands.NewItemPropertyCommand

PS C:\> Complete-Transaction
Complete-Transaction : Cannot commit transaction. The transaction has been
rolled back or has timed out.
At line:1 char:1
+ Complete-Transaction
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Complete-Transaction],
TransactionAbortedException
+ FullyQualifiedErrorId :
System.Transactions.TransactionAbortedException,Microsoft.PowerShell.Commands.
CompleteTr
ansactionCommand

PS C:\> Get-ItemProperty .
Datum : 5. 11. 2013 7:48:38
PSPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\My
Key\Datum
PSParentPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Makovec\Tran\My
Key
PSChildName : Datum
PSDrive : HKCU
PSProvider : Microsoft.PowerShell.Core\Registry

```

Všimněte si, že při snaze o vložení aktuálního data do neexistující cesty PowerShell zhlásil chybu a při snaze o zapsání celé transakce nás informoval, že nemůže provést celou transakci. Kontrolou na konci jsme si ověřili, že nová položka skutečně nebyla vytvořena.

Transakce jsou velice zajímavou součástí práce s PowerShellem, ale doufám, že se dočkáme doby, kdy budou implementovány minimálně ještě pro souborový systém. Do té doby doporučuji při vaší práci s registrem myslet na to, že je můžete využít. Za ten pocit jistoty to určitě stojí.

VYLEPŠENÝ WHERE-OBJECT

Rovnou na začátku říkám, že název článku je trochu zavádějící. Nebudeme se dnes bavit o cmdletu Where-Object, ale o jeho trochu skryté náhradě. Pro použití následujících technik musíte mít nainstalován PowerShell v4.

Před několika dny jsem se vrátil z MVP Summitu v Redmodu, kde jsem strávil několik dní v přítomnosti dalších MVPků z různých produktových skupin. V rámci mé PowerShell oblasti byly nejzajímavější přednášky na témata již existujících funkcionalit. Několik z nich jsme ale dostali rozebrané do naprostého detailu. V rámci prezentace o DSC (Desired State Configuration) – již jsem o tomto tématu [psal](#) – jsme se dostali k zajímavé vlastnosti, o kterou bych se s vámi dnes rád podělil.

V rámci konfigurace a výběru jednotlivých nodů můžete určovat, kde se bude nastavení aplikovat. Vezměme si následující příklad:

Configuration CloudService

```
{
    Node $AllNodes.Where("Role -eq Web").NodeName {
        WindowsFeature IIS
        {
            Ensure = "Present"
            Name = $Node.RolesToBePresent
        }
    }
}
```

Důležitý je pro nás třetí řádek, kde říkáme, že ze všech dostupných nodů vyberu pomocí **where** pouze ty, které mají mít roli web a z těchto si zapamatuji jejich jméno. Zajímavá je právě konstrukce **.Where(„Role -eq Web“)**. Tuto konstrukci můžeme použít i v konzoli, např.:

```
PS C:\> (Get-Process).Where({$_.name -eq "svchost"})
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
397	14	3576	9984	45	0.27	592	svchost
327	14	2384	5852	28	0.19	620	svchost
418	32	9100	13172	638	6.25	672	svchost
497	21	11368	15796	70	1.08	732	svchost
1128	45	12876	27032	192	3.56	764	svchost
462	26	4588	11056	82	0.28	800	svchost
522	33	6908	15024	1129	0.36	872	svchost
352	32	8724	10844	55	0.23	988	svchost
681	26	62800	77052	191	4.95	1224	svchost
252	15	2512	7340	47	0.06	1324	svchost

Pro porovnání, toto je standardní zápis předchozí konstrukce:

```
PS C:\> Get-Process | where name -eq 'svchost'
PS C:\> Get-Process | where { $_.name -eq 'svchost' }
```

První z nich lze použít od verze 3. Jedná se o tzv. zjednodušenou syntaxi. Poslední zápis funguje již od první verze PowerShellu. Jen pro zajímavost. Pokud bych potřeboval předchozí akci provést narychlo v konzoli, asi bych použil následující zápis:

```
PS C:\> gps|? name -eq svchost
```

Vidíte, že kolik adminů, tolik různých zápisů. Nicméně ...

Na první pohled asi vypadá nová syntaxe trochu divná. V DSC má své opodstatnění, ale v konzoli se jedná o vůbec nejdelší zápis. Výhodou je ovšem rychlost běhu.

Uložíme si všechny tři varianty do proměnné:

```
PS C:\> $c = '(gps).Where({$_.name -eq "svchost"})','gps | where { $_.name -eq "svchost" }','gps | where name -eq "svchost"'
```

Nyní si můžeme všechny příkazy spustit:

```
PS C:\> $c |% { Write-Host $_ -fore Yellow; iex $_ }  
(gps).Where({$_.name -eq "svchost"})
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
394	14	3524	9964	44	0.27	592	svchost
301	13	2216	5784	26	0.19	620	svchost
419	32	9072	13176	638	6.28	672	svchost
483	20	11312	15944	68	1.11	732	svchost
1112	45	12804	27048	191	3.69	764	svchost
429	26	4560	10784	81	0.28	800	svchost
526	33	6964	15048	1130	0.36	872	svchost
344	32	8692	10788	54	0.23	988	svchost
677	26	64500	82328	190	8.64	1224	svchost
252	15	2512	7340	47	0.09	1324	svchost

```
gps | where { $_.name -eq "svchost" }
```

394	14	3524	9964	44	0.27	592	svchost
301	13	2216	5784	26	0.19	620	svchost
419	32	9072	13176	638	6.28	672	svchost
483	20	11312	15944	68	1.11	732	svchost
1112	45	12804	27048	191	3.69	764	svchost
429	26	4560	10784	81	0.28	800	svchost
526	33	6964	15048	1130	0.36	872	svchost
344	32	8692	10788	54	0.23	988	svchost
677	28	64500	82328	190	8.66	1224	svchost
252	15	2512	7340	47	0.09	1324	svchost

```
gps | where name -eq "svchost"
```

394	14	3524	9964	44	0.27	592	svchost
301	13	2216	5784	26	0.19	620	svchost
419	32	9072	13176	638	6.28	672	svchost
483	20	11312	15944	68	1.11	732	svchost
1112	45	12804	27048	191	3.69	764	svchost
429	26	4560	10784	81	0.28	800	svchost
526	33	6964	15048	1130	0.36	872	svchost
344	32	8692	10788	54	0.23	988	svchost
677	26	64500	82328	190	8.69	1224	svchost
252	15	2512	7340	47	0.09	1324	svchost

Zde jsem si schválně dovolil použití méně známého aliasu. **iex** je alias pro **Invoke-Expression**. Tento cmdlet je schopen převzít text na vstupu a spustit jej jako příkaz PowerShellu. Ve smyčce jsem tedy prošel všechny tři varianty zápisu a pomocí **Invoke-Expression** je spustil. Pro oddělení jsem použil **Write-Host**, abychom viděli, v které fázi se nacházíme.

Nyní si pojďme porovnat rychlost všech tří variant.

```
PS C:\> $c|% { "{0,-40} : {1}" -f $_, $(Measure-Command { 1..100 |% {iex $_ } }).Milliseconds) }  
(gps).Where({$_.name -eq "svchost"}) : 46  
gps | where { $_.name -eq "svchost" } : 93  
gps | where name -eq "svchost" : 128
```

Možná bychom nejprve mohli rozklíčovat zápis. Opět procházíme všechny varianty, ale jejich výpis nyní zobrazujeme pomocí **F operátoru**. Pokud o něm chcete vědět víc, můžete se podívat [na mé staré články](#). První část zobrazuje text a druhá vlastní čas běhu. Pro měření běhu jsem využil cmdlet **Measure-Command** a poté zobrazil pouze jeho vlastnost **Milliseconds**. Je vidět, že rozdíl v běhu je docela markantní. Největší rezie je u posledního zápisu. V případě, že bychom filtrovali nějaké větší pole objektů, může nám nový zápis hodně zkrátit celkový čas skriptu.

Pojďme si tedy ukázat další možnosti daného zápisu. Where můžete řetězit za sebe. Můžete tedy vyzkoušet toto:

```
PS C:\> (gps).Where({$_.name -eq "svchost"}).Where({$_.id -ge 800})
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
458	26	4856	11300	82	0.28	800	svchost
519	32	6936	15092	1129	0.38	872	svchost
348	32	8848	10840	55	0.23	988	svchost
675	26	64504	82368	190	12.67	1224	svchost
252	15	2512	7340	47	0.13	1324	svchost

Můžete si nechat vypsat pouze první nebo poslední záznam:

```
PS C:\> (gps).Where({$_.name -eq "svchost"}, "last")
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
252	15	2512	7340	47	0.16	1324	svchost

```
PS C:\> (gps).Where({$_.name -eq "svchost"}, "first")
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
394	14	3524	9968	44	0.27	592	svchost

Zajímavé je, že ve stejném tvaru můžete použít i **foreach**. Můžete tedy zkusit:

```
PS C:\> (gps).Where({$_.name -eq "svchost"}).foreach({$_.name})
```

```
svchost
svchost
svchost
svchost
svchost
svchost
svchost
svchost
svchost
svchost
```

Kdy pro každý z objektů vypisujeme pouze jeho jméno. Samozřejmě byste asi dokázali vymyslet užitečnější příklad. Zkuste si například vzít některý z vašich posledních použitých příkazů a převést jej na tuto syntaxi. Klidně si i změřte délku běhu.

Dnešním cílem nebylo přesvědčit vás na využívání této syntaxe. Chtěl jsem vám ukázat jednu z možných cest a je na vás, abyste se rozhodli. Určitě je dobré o této variantě vědět a v případě, že budete mít pocit, že by se vám mohla hodit, vyzkoušejte ji. Nebo budete alespoň vědět, že jste to již někde viděli, pokud se v budoucnu dostanete k článku, kde bude tento zápis použit.

Ještě jednou připomínám, že pro možnost použití tohoto tvaru, musíte mít instalován PowerShell v4.

VYLEPŠENÁ KONZOLE

Dnes bych vám rád ukázal modul, který hezkým způsobem zjednodušuje práci v konzoli.

Na téma konzole bylo napsáno a řečeno mnoho (a většinou to byly nepěkné věci). Mnoho lidí se diví, proč PowerShell používá stejnou konzoli jako cmd.exe. Sám Jeffrey Snover jednou řekl, že to je dědictví minulosti, kterého se bohužel PowerShell jen tak nezbaví.

Existuje mnoho náhrad pro konzoli, namátkou můžu jmenovat např. [PoshConsole](#) nebo [ConEmu](#). Nesmíme zapomínat ani na PowerShell ISE. Já osobně nepoužívám žádný z prvně jmenovaných editorů. Hlavním důvodem je to, že si chci vystačit se základní nabídkou standardního PowerShellu.

To ovšem neznamená, že rozšíření konzole považuji za zlo. Spíše využívám nabídky různých modulů a úpravu svého profilu. Jak jsem již psal dříve – profil, skripty a moduly spouštím z adresáře, který mám synchronizovaný v cloudu a na všech svých počítačích. Tím mám vyřešeny „své“ konzole. Jedním z modulů, který takto využívám je [PSReadLine](#).

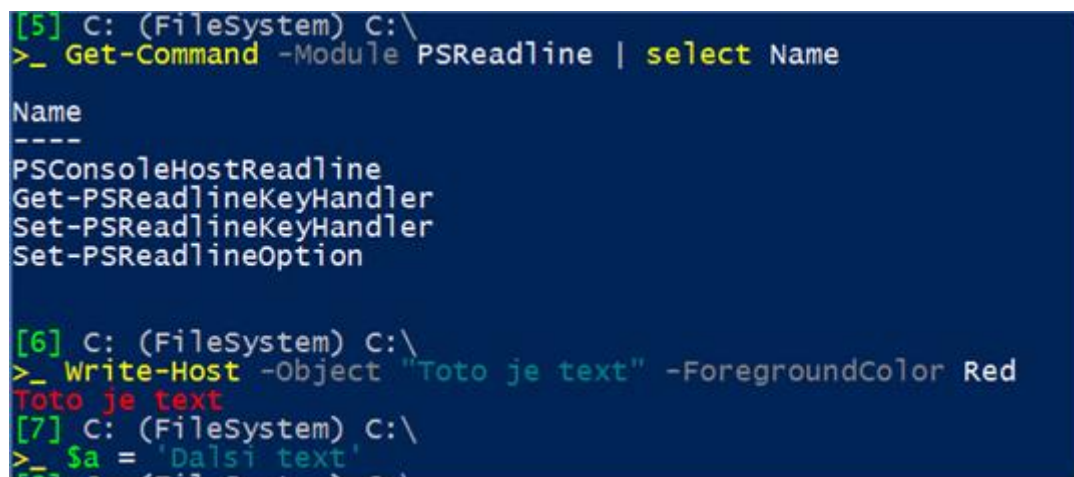
Instalace je jednoduchá, stačí si stáhnout [ZIP soubora](#) nakopírovat jej do adresáře s moduly. Poté jej můžete importovat buď ručně z konzole, nebo ze svého profilu. Modul nefunguje v PowerShell ISE, takže je potřeba při volání z profile.ps1 určit, že importujeme pouze z konzole:

```
if ($host.Name -eq 'ConsoleHost')
{
Import-Module PSReadline
}
```

Modul má několik zajímavých funkcí (některé pokročilejší jsou popsány na stránce zmiňované výše), ukážeme si ty zřejmě nejčastěji využívané.

Zvýrazňování syntaxe

Na první pohled poznáte nejviditelnější změnu: zvýrazňování syntaxe. Funguje jak na cmdlety, tak na proměnné, texty, ...



```
[5] C: (FileSystem) C:\
> Get-Command -Module PSReadline | select Name

Name
----
PSConsoleHostReadline
Get-PSReadlineKeyHandler
Set-PSReadlineKeyHandler
Set-PSReadlineOption

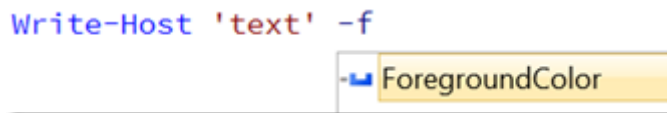
[6] C: (FileSystem) C:\
> Write-Host -Object "Toto je text" -ForegroundColor Red
Toto je text
[7] C: (FileSystem) C:\
> $a = 'Další text'
```

Jedná se o velice jednoduchou pomůcku, ale co se týče přehlednosti, nemůžu si ji vynachválit.

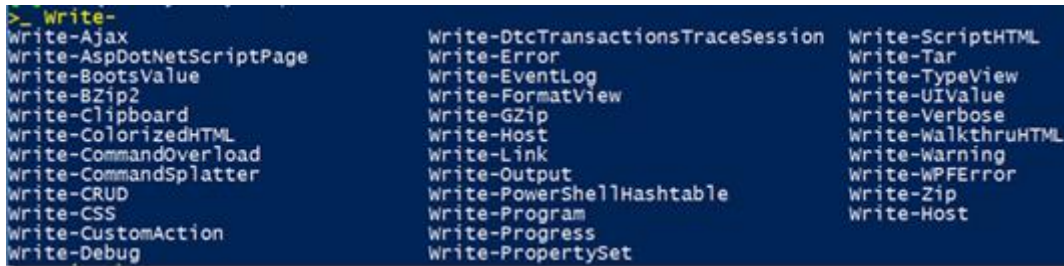
Doplňování

Další zajímavostí je doplňování příkazů a parametrů. Standardně můžete několikanásobným stiskem klávesy Tab vyvolat doplňování např. jmen cmdletů. V konzoli ovšem nefunguje Intelli Sense ve smyslu, že vidíte všechny příkazy, parametry, které splňují „prozatímní zadání“. PsReadLine vám toto umožňuje.

V ISE jistě znáte toto:



Pokud Intelli Sense zmizí, můžete jej vyvolat zpětně kombinací kláves Ctrl+mezera. To samé vám umožňuje PsReadLine v konzoli. Ukažme si to prakticky. Po napsání **Write**-jsem stiskl Ctrl+mezera:

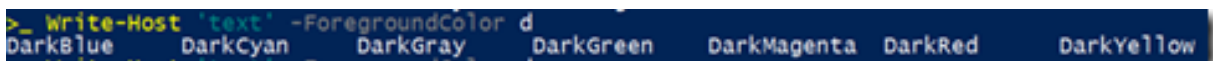


V konzoli se objeví seznam cmdletů splňujících mé zadání.

Stejný postup použijí i na dalších obrázcích. Zde vidím jméno parametru



A zde hodnoty parametru, které mohu použít:



Nejčastěji tuto funkcionalitu využívám, když se chci podívat, jak cmdlety jsou dostupné. Standardním způsobem je volání Get-Command:

```
>_ Get-Command -Module Azure -Verb get
```

CommandType	Name	ModuleName
-----	-----	-----
Cmdlet	Get-AzureAccount	Azure
Cmdlet	Get-AzureAclConfig	Azure
Cmdlet	Get-AzureAffinityGroup	Azure
Cmdlet	Get-AzureCertificate	Azure
Cmdlet	Get-AzureDataDisk	Azure
Cmdlet	Get-AzureDeployment	Azure
Cmdlet	Get-AzureDisk	Azure
Cmdlet	Get-AzureDns	Azure
Cmdlet	Get-AzureEndpoint	Azure
Cmdlet	Get-AzureEnvironment	Azure
Cmdlet	Get-AzureLocation	Azure
Cmdlet	Get-AzureMediaServicesAccount	Azure
Cmdlet	Get-AzureOSDisk	Azure
Cmdlet	Get-AzureOSVersion	Azure
Cmdlet	Get-AzurePublishSettingsFile	Azure
Cmdlet	Get-AzureRemoteDesktopFile	Azure
Cmdlet	Get-AzureRole	Azure
Cmdlet	Get-AzureSBAuthorizationRule	Azure
Cmdlet	Get-AzureSBLocation	Azure
Cmdlet	Get-AzureSBNamespace	Azure
Cmdlet	Get-AzureService	Azure
... (zkráceno)		

Obrazovka mi odjede a já pak skroluji zpět, abych se podíval. Mohu samozřejmě využít cmdlet Format-Wide pro jednodušší zobrazení:

```
> Get-Command -Module Azure -Verb get | fw -c 3
Get-AzureAccount           Get-AzureAclConfig         Get-AzureAffinityGroup
Get-AzureCertificate       Get-AzureDataDisk         Get-AzureDeployment
Get-AzureDisk              Get-AzureDns              Get-AzureEndpoint
Get-AzureEnvironment       Get-AzureLocation          Get-AzureMediaServicesAccount
Get-AzureOSDisk            Get-AzureOSVersion         Get-AzurePublishSettingsFile
```

Ale pokud vím, že tým, který cmdlety vytvářel, používá standardní jmennou konvenci a prefix pro své cmdlety, mohu použít PsReadLine:

```
> get-azure
Get-AzureAccount           Get-AzureSBNamespace       Get-AzureStorageTable
Get-AzureAclConfig        Get-AzureService           Get-AzureStoreAddOn
Get-AzureAffinityGroup    Get-AzureServiceDiagnosticsExtension Get-AzureSubnet
Get-AzureCertificate       Get-AzureServiceProjectRoleRuntime Get-AzureSubscription
Get-AzureDataDisk         Get-AzureServiceRemoteDesktopExtension Get-AzureTable
Get-AzureDeployment        Get-AzureSqlDatabase        Get-AzureVM
Get-AzureDisk             Get-AzureSqlDatabaseImportExportStatus Get-AzureVMImage
Get-AzureDns              Get-AzureSqlDatabaseServer  Get-AzureVNetConfig
```

Tipy

Po instalaci modulu bych vám doporučil vyzkoušet si ukázané příklady. Poté zjistíte, že byste vaši konzoli chtěli dále upravovat.

Příkaz, který zatím standardně v PsReadLine chybí je Ctrl+End – mazání řádky až do konce. Tuto funkcionalitu můžete doplnit pomocí následujícího příkazu:

```
Set-PSReadlineKeyHandler -Key Ctrl+End -BriefDescription KillLine -Handler {
[PSConsoleUtilities.PSConsoleReadLine]::KillLine() }
```

Další typy jsou dostupné na již zmíněné stránce. Abych jenom nechválil. Jedna věc mi na PsReadLine vadí. Při kopírování většího množství textu do konzole (typicky listing funkce, kterou chci vyzkoušet) probíhá vložení do konzole velmi pomalu (efekt psacího stroje). Je to tím, že probíhá kontrola syntaxe při vložení každého znaku a zpomalení je v některých případech znatelné.

Doufám, že se vám dnešní tip líbil a že vám přijde vhod.

CYKLOTOULKY

Dneska se trošku projedeme. Ale nebojte se, nemusíte vytáhnout paty z konzole, resp. ISE.

Chtěl bych vám dnes ukázat, jaké možnosti máme v PowerShellu při práci s cykly. Jelikož jsem v poslední době viděl několik špatných použití různých cyklovacích algoritmů.

Všechny následující příklady budu ukazovat na stejném příkazu. Mým cílem bude vypsat jména běžících procesů. Výsledek by měl být vždy stejný jako v následujícím případě

```
Get-Process | Select Name
```

For

Jedná se o základní příkaz cyklu, který existuje snad ve všech programovacích jazycích. Můžete jej použít v případě, kdy máte jasně daný počet opakování cyklu.

```
$procesy = Get-Process
for ($i = 0; $i -lt $procesy.Count; $i++)
{
    $procesy[$i].Name
}
```

Do proměnné \$procesy jsme si uložili všechny procesy. Poté v cyklu **for** použijeme proměnnou \$i, která nabývá hodnot od 0 do počtu procesů. Pozor na to, že pole jsou v PowerShellu číslována od nuly a proto je první člen z proměnné \$procesy indexován nulou.

Předchozí kód můžeme tedy číst následovně: *Ulož procesy do proměnné. Do proměnné \$i ulož nulu. Vrať proces s indexem 0 a zobraz jeho jméno. Proměnnou \$i zvyš o 1 (řečeno v \$i++). Proved' znovu cyklus – vrať proces s indexem 1. Vše opakuj, až do počtu procesů uložených v proměnné \$procesy.*

While

Cyklus probíhá tak dlouho, dokud je splněna podmínka. Tedy

```
$j = 0
while ($j -le $procesy.Count)
{
    $procesy[$j].Name
    $j++
}
```

Předpokládáme proměnnou \$procesy. Nastavíme \$j na nulu. Podmínka nám říká, že dokud je \$j menší nebo rovnou počtu procesů, procesy vypisujeme. Zároveň zvyšujeme \$j o jednotku po každém výpisu.

Do

Tento příkaz se hodně často uveden jako „dvojče“ předchozího. Ukážeme si rovnou použití

```
$k = 0
do
{
    $procesy[$k].Name
    $k++
}
while ($k -le $procesy.Count)
```

Jak vidíte, rozdíl je použití podmínky na konci cyklu. U použití **Do** proběhne celý cyklus alespoň jednou. Což je rozdíl oproti použití předchozího příkazu **while**.

Příkaz **Do** lze použít ještě s klíčovým slovem **until**. V tomto případě je potřeba obrátit závěrečnou podmínku.

```
$k = 0
do
{
    $procesy[$k].Name
    $k++
}
until ($k -ge $procesy.Count)
```

Přečtěte si pozorně oba předchozí příklady a snažte se opravdu pochopit, jak fungují.

Všechny předchozí příklady jsem uvedl proto, že jsem dnes chtěl hlavně mluvit o asi nejpoužívanějším příkazu cyklu: **foreach**. Vzhledem k tomu, že jsem nedávno viděl špatné použití, rád bych se u něj zastavil na delší čas.

ForEach

Tento příkaz používejte vždy, když máte danou kolekci objektů a chcete je projít všechny. Typicky při použití Get* cmdletů a procházení výsledků. V našem případě tedy.

```
foreach ($p in $procesy)
{
    $p.Name
}
```

Zde vidíte úplně jiný přístup než v předchozích příkladech. Můžeme jej číst takto. *Pro každý proces v kolekci uložených procesů (proměnná \$procesy) vypiš jeho jméno.*

Uvědomte si jednu důležitou vlastnost. V rámci **foreach** pracujete s proměnnou \$p, nikoli \$procesy! Pokud byste použili proměnnou \$procesy uvnitř cyklu, dostanete nepředvídatelné výsledky.

Často můžete tento příklad vidět zapsaný i takto:

```
foreach ($p in Get-Process)
{
    $p.Name
}
```

Zde neukládáme mezi výsledky do proměnné, ale rovnou uvedeme Get-Process, který nám do cyklu **foreach** posílá jednotlivé procesy.

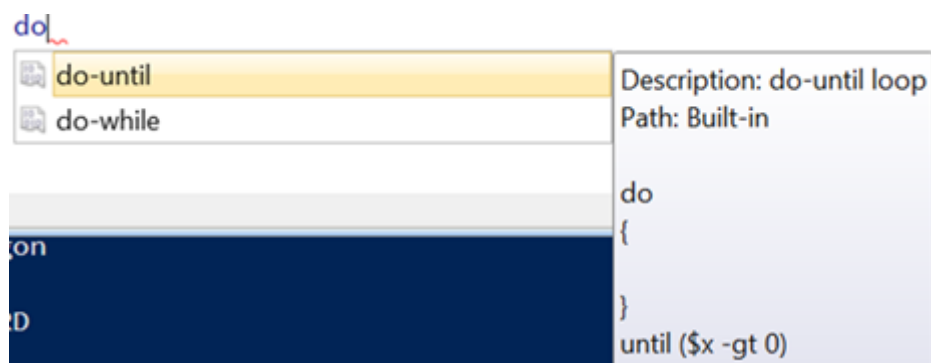
Pokud se podívám na své skripty, můžu říci, že v 99% případů používám **foreach**. Důvod je prostý – použití v IT Pro světě je ve většině případů typu: dej mi všechny položky a s každou z nich něco udělej. Ať se jedná o procesy, služby, soubory, virtuální počítače (VMM), kolekce (ConfigMgr), ...

Závěrem

Pokud se chcete o zde zmíněných příkazech dozvědět více, podívejte se do nápovědy. Témata, která vás budou zajímat, jsou:

- about_for
- about_while
- about_do
- about_foreach

Jednou z věcí, která vám při používání pomůže, je ISE. Od verze 3 můžete použít tzv. Snippets. V ISE stisknete Ctrl+J a poté napíšete jméno cyklu, který vás zajímá.

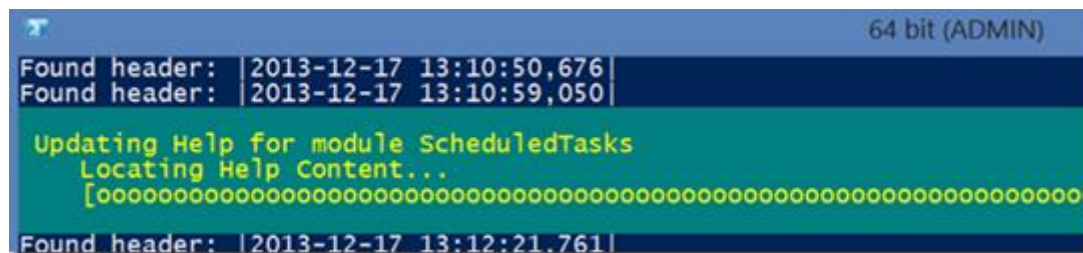


Poté, co stisknete klávesu Tab, ISE vám doplní strukturu příkazu. Na vás je již jen doplnit správné proměnné.

A ještě jedna poznámka z praxe. Od PowerShellu v3 máte možnost aktualizovat nápovědu. Rozhodně doporučuji tuto možnost využít. Není nic horšího, když po vás klient chce pomoci s modulem, o

kterém jste nikdy neslyšeli. Nápověda je prvním zdrojem, po kterém vždy sahám. Jak jsem již psal několikrát – pokud znáte základy používání PowerShellu, není pro vás problém nasadit je na jakýkoli modul. Vždy je ale potřeba vědět, jak se chovají jednotlivé cmdlety. Pokud nemáte nápovědu, vše je o něco složitější.

Proto doporučuji pravidelně používat **Update-Help**.



Po doběhnutí celého procesu budete moci zkoumat cmdlety přímo z konzole.

ZÁCHRANNÉ PARAMETRY

Dnešní článek je opět inspirován životem a drsnou zkušeností. Ale nemusíte se bát, žádnému zvířátku se nic nestalo a nebohý administrátor o práci nepřišel.

Mám jedno pravidlo, které někteří „odvážnější“ kolegové a zákazníci občas se smíchem znevažují. **Vždycky** když pouštím nějaký skript či one-liner na produkčním serveru, používám parametr **WhatIf**. Zároveň mám rád, když při prvních testech mohu použít parametr **Verbose**. Chci prostě vidět, co PowerShell dělá a mít šanci v dalším běhu zasáhnout. Myslím, že jsem tuto informaci už několikrát zmiňoval. Pojdme se podívat na nejčerstvější příběh.

Administrátor jedné firmy (říkejme mu Tomáš, Kanty s dovolením promine) vytvářel virtuální počítače pomocí jednoho cmdletu. Jelikož potřeboval vše vyřešit rychle, našel jeden takový na webu, vložil jej do konzole, a hle – než si všiml, co se děje, smazal si několik virtuálů ze svého Hyper-V serveru.

Určitě to znáte, na webu vidíte kód, který se nevejde na šířku stránky, takže tento kód pokračuje a vy nevidíte, co je dále. Ale to přeci nevadí, označíte celou řádku, vložíte ji do konzole a pak upravíte. Pozor! Zde je jedna nepříjemná vlastnost konzole – konec řádku je Enter a Enter spouští příkaz. To, co Tomáš viděl na webu, bylo zhruba (teoretický příklad, syntaxi si již nepamatuji):

```
Get-VM -Name * | Set-VM -Network něco -Memory 12
```

Ovšem co neviděl za koncem sloupce bylo:

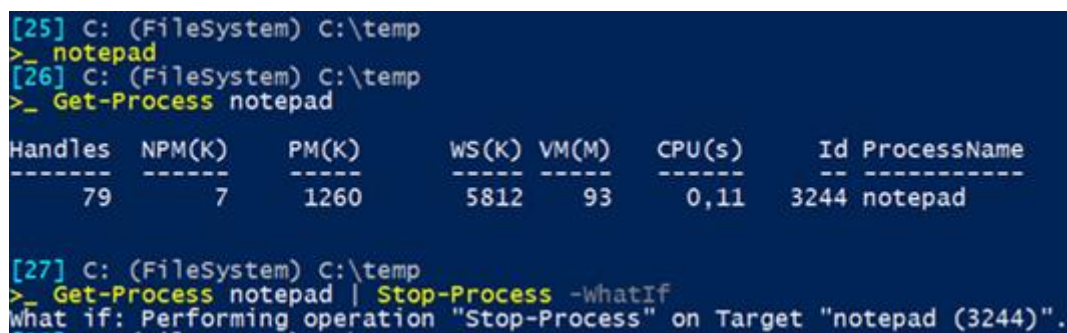
```
... | Remove-VM
```

Opravdu teď nedokážu z hlavy vydolovat, proč na webu řádka končila odebráním počítače, ale to teď není důležité. Výsledek je jasný – mizející seznam počítačů z Hyper-V konzole.

Takže od té doby je vidět, že každý kód, který kopíruje, jde nejdříve do notepadu a teprve poté do konzole. Stejnou službu provede i ISE, které ve skriptu kód nespouští.

Jak můžete něčemu takovému zamezit? Mým oblíbeným je WhatIf. Před několika lety jsem kdesi popisoval, jak zapnout natrvalo tento parametr. Nyní můžeme použít mou oblíbenou proměnnou **PsDefaultParameterValues**.

Pro ty, co neznají WhatIf. Zkuste si pustit následující příkazy:

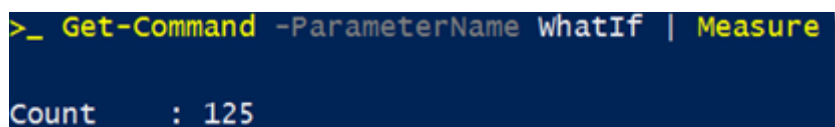


```
[25] C: (FileSystem) C:\temp
> notepad
[26] C: (FileSystem) C:\temp
> Get-Process notepad

Handles      NPM(K)      PM(K)      WS(K) VM(M)      CPU(s)      Id ProcessName
-----
79           7          1260       5812    93         0,11      3244 notepad

[27] C: (FileSystem) C:\temp
> Get-Process notepad | Stop-Process -WhatIf
What if: Performing operation "Stop-Process" on Target "notepad (3244)".
```

Pokud bych vynechal WhatIf, proces by byl zastaven. Stejně jako začaly být odebírány virtuální počítače na začátku tohoto článku. Tímto ovšem nezamezíme plně zmiňovanému scénáři kopírování z webu – WhatIf bychom nestačili dopsat. Podívejme se, kolik cmdletů WhatIf používá.



```
> Get-Command -ParameterName WhatIf | Measure

Count      : 125
```

Když se podíváme na to, o jaký typ cmdletů se jedná:

```
> Get-Command -ParameterName WhatIf | Group-Object Verb | Sort-Object Count -Desc | Select-Object -First 5 | Format-Table Name, Count -Auto
```

Name	Count
Remove	28
Set	16
New	11
Clear	6
Enable	4

Je nejčastější případ Remove.

Proměnná WhatIfPreference určuje, že ve standardním režimu není WhatIf zapnuté. Pomocí nastavení na True se budou cmdlety chovat, jako by tento parametr byl vždy uveden. Pomocí PsDefaultParameterValues uděláme to samé, ale cestou, kterou pravděpodobně nastavujeme i další chování různých kombinací cmdletů a jejich parametrů.

```
[35] C: (FileSystem) C:\temp
> $PSDefaultParameterValues.Add('*:WhatIf', $true)
[36] C: (FileSystem) C:\temp
> notepad
[37] C: (FileSystem) C:\temp
> Get-Process notepad | Stop-Process
What if: Performing operation "Stop-Process" on Target "notepad (2488)".
```

Vidíte, že i bez uvedení parametru WhatIf se notepad nezavřel. Pokud bychom chtěli tuto funkcionalitu vyřadit, musíme určit hodnotu parametru. U přepínačů to probíhá následujícím způsobem:

```
[39] C: (FileSystem) C:\temp
> Get-Process notepad | Stop-Process -WhatIf:$false
```

V tomto případě je již notepad vypnut. Nevýhodu vidím v tom, že proměnná WhatIfPreference je stále nastavena na False:

```
> $WhatIfPreference
False
```

Což může být pro někoho matoucí. Můžete si vybrat libovolnou z cest a potřebné nastavení si vložit do svého profilu.

Podobné to je s parametrem Verbose. Pokud chcete, aby se vám podrobné informace vždy zobrazovaly, nastavte si v profilu hodnotu tak, jak potřebujete. Více informací o Verbose a jeho chování najdete v helpu, např.:

```
> help verbosepreference
```

Name
Write-Verbose
about_CommonParameters
about_Preference_Variables
about_Variables

Mimochodem, všimli jste si, jak je konzole při použití PSReadLine čitelnější? Více jsem o tomto modulu psal v jednom z předchozích Flashů. Jestli jste ještě nezkoušeli, vyzkoušejte je

NOVINKY VERZE 5

Dobrý den. Vítám vás u nového Flashe, nového článku a novinek ze světa PowerShellu. Dnes nás čeká trochu oddychový, ale o to příjemnější – alespoň pro mne – článek.

E+T v Praze

Prvním skvělým zážitkem uplynulého měsíce byla návštěva Eda Wilsona AKA Scripting Guy a jeho ženy Teresy v Praze. A to nejen z pohledu Edovy prezentace na téma Workflows a DSC (obě jsou již

dostupné na MSTV.cz), ale i z pohledu osobního setkání a možnosti mít oba dva „pro sebe“ na celý týden. Z celého setkání vzešlo několik zajímavých podnětů a s některými vás již brzy seznámím.

Musím říci, že oba dva si pobyt zde užili a byli příjemně překvapeni reakcí lidí po Edově přednášce, množstvím dotazů a přátelskou atmosférou.

Aprílová MVP nominace

Vždy čtyřikrát za rok rozesílá Microsoft maily všem novým a „obnoveným“ MVP. Jedním z termínů je i první duben. Zatím si nikdo nedovolil o své případné nominaci žertovat a tak se stalo i letos. S napětím jsme letos čekali, jestli dojde k dalšímu průlomu pro ČR a povedlo se. S radostí musím oznámit, že [Jakub Jareš](#) se stal dalším českým PowerShell MVP. Jakub je velice aktivním „odpovídačem“ v PowerShell fórech a mimo jiné napsal i několik skvělých článků pro [PowerShellMagazine](#). Na Twitteru jej můžete sledovat pod jeho nickem [nohandle](#).

Dalším členem rodiny PowerShell MVP se stala také Teresa Wilson AKA Scripting Wife. Jsem velmi rád, že Teresy práce pro komunitu byla oceněna nejvyšším možným způsobem. Její organizační schopnosti pomohly již na mnoha konferencích a v mnoha PowerShell User Groups.

Novinky: WMF 5 Preview

Se zkrácením cyklu pouštění nových OS, se také zrychlil přísun novinek v PowerShellu. Jak určitě víte, PowerShell je součástí balíčku nazvaného Windows Management Framework. Tento Framework byl nyní uvolněn ve verzi 5 pro testování. Jedná se zatím jen o preview, ale alespoň je naznačen směr vývoje a možná největší novinka ostré páté verze.

Pokud si chcete WMF5 nainstalovat, je dostupný se stažení na <http://www.microsoft.com/en-us/download/details.aspx?id=42316>. V současné době je dostupný pro Windows 8.1 a Windows Server 2012 R2. Zatím rozhodně nedoporučuji instalovat WMF na produkční servery. Stejně jako v případě vydání WMF4 jsou oznámeny nekompatibility s některými nástroji. Pro jejich seznam se podívejte do sekce System Requirements na zmiňované stránce.

První novinkou je uvolnění cmdletů pro správu switchů. A teď nemyslím switche v PowerShellu, ale opravdové, fyzické síťové switche. Samozřejmě se musí jednat o switche, které budou certifikované. Na loňském MVP Summitu nám byla ukázána jedna z prvních verzí a jsem rád, že PowerShell se rozšiřuje tímto směrem. Seznam dostupných příkazů si můžeme vypsát standardním způsobem:

```
PS C:\Users\makovec> Get-Command -Module NetworkSwitch
```

CommandType	Name	Source
-----	----	-----
Function	Disable-NetworkSwitchEthernetPort	NetworkSwitch
Function	Disable-NetworkSwitchFeature	NetworkSwitch
Function	Disable-NetworkSwitchVlan	NetworkSwitch
Function	Enable-NetworkSwitchEthernetPort	NetworkSwitch
Function	Enable-NetworkSwitchFeature	NetworkSwitch
Function	Enable-NetworkSwitchVlan	NetworkSwitch
Function	Get-NetworkSwitchEthernetPort	NetworkSwitch
Function	Get-NetworkSwitchFeature	NetworkSwitch
Function	Get-NetworkSwitchGlobalData	NetworkSwitch
Function	Get-NetworkSwitchVlan	NetworkSwitch
Function	New-NetworkSwitchVlan	NetworkSwitch
Function	Remove-NetworkSwitchEthernetPortIPAddress	NetworkSwitch
Function	Remove-NetworkSwitchVlan	NetworkSwitch
Function	Restore-NetworkSwitchConfiguration	NetworkSwitch
Function	Save-NetworkSwitchConfiguration	NetworkSwitch
Function	Set-NetworkSwitchEthernetPortIPAddress	NetworkSwitch
Function	Set-NetworkSwitchPortMode	NetworkSwitch
Function	Set-NetworkSwitchPortProperty	NetworkSwitch
Function	Set-NetworkSwitchVlanProperty	NetworkSwitch

Druhou novinkou, dle mého pro administrátory zajímavější, je systém pro správu balíčků. Např. pro vývojáře se jedná o velice známý mechanismus. Pro jednoduchost si představte systém, kdy máte repository se softwarem a pomocí několika příkazů můžete tento software instalovat. Právě jsem si

tento systém pojmenoval jako ConfigMgr pro příkazovou řádku Ale neberte mne tak úplně vážně.

Novinku Microsoft nazval **OneGet**. Pojd'me si ji ukázat v praxi. OneGet je dodáván jako modul:

```
PS C:\Users\makovec> gcm -Module oneget
```

CommandType	Name	Source
-----	----	-----
Cmdlet	Add-PackageSource	oneget
Cmdlet	Find-Package	oneget
Cmdlet	Get-Package	oneget
Cmdlet	Get-PackageSource	oneget
Cmdlet	Install-Package	oneget
Cmdlet	Remove-PackageSource	oneget
Cmdlet	Uninstall-Package	oneget

Pro správnou funkcionalitu je zapotřebí mít systém, kde jsou balíčky (konkrétní software) uloženy. OneGet používá systém [Chocolatey](#). Seznam zdrojů lze rozšiřovat a v rámci organizace si například vytvořit vlastní zdroj, který bude např. jediný dostupný pro interní IT.

```
PS C:\Users\makovec> Get-PackageSource | ft -au
```

Name	Location	Provider	IsTrusted
----	-----	-----	-----
chocolatey	http://chocolatey.org/api/v2/	Chocolatey	False

Všimněte si vlastnosti **IsTrusted**– zdroj je nastaven jako nedůvěryhodný. Toto můžete změnit. Pokud necháte nastavení jak je, při instalaci balíčku se objeví varování, které musíte pro instalaci potvrdit.

Dostupné balíčky zobrazíte pomocí **Find-Package**.

```
PS C:\Users\makovec> Find-Package
```

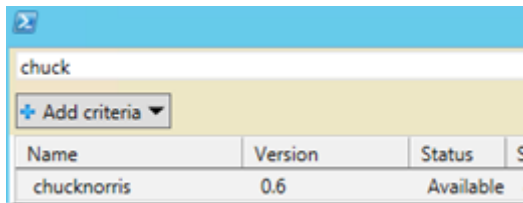
Name	Version	Status	Source	Summary
----	-----	-----	-----	-----
1password	1.0.9.340	Available	chocolatey	1Password - Have
you ever forgotte...				
7zip	9.22.01.20130618	Available	chocolatey	7-Zip is a file
archiver with a hi...				
7zip.commandline	9.20.0.20130618	Available	chocolatey	7-Zip is a file
archiver with a hi...				
7zip.install	9.22.01.20130618	Available	chocolatey	7-Zip is a file
archiver with a hi...				
ack	2.04	Available	chocolatey	ack is a tool like
grep, designed ...				
acr	2.6.0	Available	chocolatey	
ActivePerl	5.14.2.2	Available	chocolatey	ActivePerl is the
leading commerci...				
ActiveTcl	8.5.14.002	Available	chocolatey	
ActySystemStyleRule	1.0.0	Available	chocolatey	
ad-awarefreeantivirus	11.1.5354.0	Available	chocolatey	
adblockpluschrome	0.0.0.2	Available	chocolatey	
adblockplusfirefox	0.0.0.1	Available	chocolatey	
adblockplusie	1.1	Available	chocolatey	
adblockplusopera	0.0.0.1	Available	chocolatey	

addtopath 1.0 Available chocolatey Adds a right click option on windo...
(zkráceno ...)

```
PS C:\Users\makovec> (Find-Package | Measure).Count
1741
```

```
PS C:\Users\makovec> Find-Package | Out-GridView
```

Nyní je dostupných celkem 1741 balíčku (počet se mění každým dnem). Pro prozkoumání doporučuji přeměřovat seznam do Out-GridView. Z následujícího obrázku je vidět, že Tomáš by se mohl dočkat [svého dalšího snu](#):



Po nalezení balíčku jej můžeme nainstalovat. Na každý server například instaluji 7-Zip:

```
PS C:\Users\makovec> Find-Package -Name 7zip
```

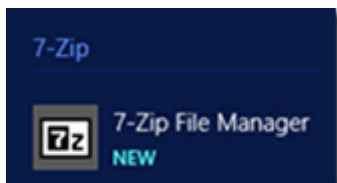
Name	Version	Status
7zip	9.22.01.20130618	Available

```
PS C:\Users\makovec> Install-Package -Name 7zip
```

```
Installing Package '7zip' from untrusted source
WARNING: This package source is not marked as safe. Are you sure you want to
install software from 'chocolatey'
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

Name	Version	Status	Source	Summary
7zip.install	9.22.01.20130618	Installed	chocolatey	7-Zip is a file archiver with a hi...
7zip	9.22.01.20130618	Installed	chocolatey	7-Zip is a file archiver with a hi...

Jak vidíte, musel jsem potvrdit, že chci instalovat 7Zip z nedůvěryhodného zdroje. Ihned po instalaci je 7-Zip dostupný a lze jej používat. Ve Start nabídce je vidět, že jsem instaloval nový software:



Osobně si myslím, že toto bude hlavní novinkou PowerShell v5. Vede mne k tomu ještě jeden důvod:

```
PS C:\Users\makovec> Get-DscResource
```

ImplementedAs	Name
Binary	File
PowerShell	Archive
PowerShell	Environment
PowerShell	Group
Binary	Log

PowerShell	Package
PowerShell	Registry
PowerShell	Script
PowerShell	Service
PowerShell	User
PowerShell	WindowsFeature
PowerShell	WindowsProcess
PowerShell	xOneGet

Všimněte si poslední položky. Pokud jste byli na [Edově přednášce o DSC](#), možná je vám to již jasné. Jelikož je OneGet uvolněn i jako zdroj do DSC, můžete jej v DSC použít. Představte si vytvoření počítače přes DSC a následnou instalaci softwaru přes OneGet. Jedná se o skvělou kombinaci. Zatím se jde pouze o experimentální verzi, ale opět vidíme světlou budoucnost.

//build/

Na právě skončené [konferenci Build](#), bylo uvedeno několik dalších novinek. Pro mne osobně největšími novinkami jsou ty, spojené s [Microsoft Azure](#). Vzhledem k novinám byla uvolněna i nová verze PowerShell modulu pro správu Azure. Ale o správě a novinkách v PowerShellu ve vztahu k Azure si povíme někdy příště.

PŘEJÍMÁNÍ SKRIPTŮ Z INTERNETU

nes bych rád navázal na jeden z posledních článků, kde jsem mluvil o používání WhatIf. Byl jsem požádán o pomoc při řešení problému se staženým skriptem – nefungoval. Rovnou říkám, že dotaz byl od běžného IT admina – o PowerShellu má základní povědomost, ale vždy si vystačí s vyhledáním řešení na internetu. To, co najde, poté vloží do konzole nebo ISE, spustí to, a doufá, že se něco (dobrého) stane. Osobně si nedokážu představit, co by se stalo, kdyby mu někdo podstrčil destruktivní kód. Ale to už je jiný příběh.

Původní soubor, ke kterému jsem se dostal, obsahoval následující kód:

```
# convert an octet string guid to a typical string GUID
function Convert-OctetStringToGuid
{
    param
    (
        [String]$Guid
    );
    if(32 -eq $guid.Length)
    {
        [UInt32]$a = [Convert]::ToUInt32(($guid.Substring(6, 2) +
$guid.Substring(4, 2) + $guid.Substring(2, 2) + $guid.Substring(0,
2)), 16)
        [UInt16]$b = [Convert]::ToUInt16(($guid.Substring(10, 2) +
$guid.Substring(8, 2)), 16)
        [UInt16]$c = [Convert]::ToUInt16(($guid.Substring(14, 2) +
$guid.Substring(12, 2)), 16)
        [Byte]$d = ([Convert]::ToUInt16($guid.Substring(16, 2), 16)
-as [byte])
        [Byte]$e = ([Convert]::ToUInt16($guid.Substring(18, 2), 16)
-as [byte])
        [Byte]$f = ([Convert]::ToUInt16($guid.Substring(20, 2), 16)
```

```

-as [byte])
    [Byte]$g = ([Convert]::ToUInt16($guid.Substring(22, 2), 16)
-as [byte])
    [Byte]$h = ([Convert]::ToUInt16($guid.Substring(24, 2), 16)
-as [byte])
    [Byte]$i = ([Convert]::ToUInt16($guid.Substring(26, 2), 16)
-as [byte])
    [Byte]$j = ([Convert]::ToUInt16($guid.Substring(28, 2), 16)
-as [byte])
    [Byte]$k = ([Convert]::ToUInt16($guid.Substring(30, 2), 16)
-as [byte])
    [Guid]$g = New-Object Guid($a, $b, $c, $d, $e, $f, $g, $h,
    $i, $j, $k)
    return $g.Guid;
}
else
{
    throw Exception('Input string is not a valid octet string
GUID')
}
}

```

Soubor byl nazván *Convert-OctetStringToGuid.ps1* a při spuštění samozřejmě neprovedl žádnou akci. Předpokládám, že čtenáři Flashe ví proč. Při spuštění skriptu proběhl vnitřní kód, ale ten pouze vytvořil funkci, která nebyla dále spuštěna. Situace má dvě řešení

1. Do skriptu vložit i volání funkce s daným parametrem.
2. Zavolat skript pomocí tzv. dot-sourcingu, kdy funkce zevnitř skriptu přeneseme do aktuálního prostředí. O dot-sourcingu jsem již psal a i v první PowerShell akademii je o něm zmínka. Skript by se tedy spustil následujícím způsobem:

```
PS> . ./Convert-OctetStringToGuid.ps1
```

Ano, první tečka je opravdu správně, to je onen dot-sourcing. Řekněme, že jsme jej použili a funkci máme nyní dostupnou v konzoli.

Vstupem je GUID z Exchange. Ten je standardně ve tvaru: *d05db2e2-42de-4998-80cc-90fdc8111c54*. Takže jej zkusíme zadat jako parametr naší funkce:

```
PS C:\> Convert-OctetStringToGuid d05db2e2-42de-4998-80cc-90fdc8111c54
```

```

Exception : The term 'Exception' is not recognized as the name of a
cmdlet, function, script file, or operable program.
Check the spelling of the name, or if a path was included, verify
that the path is correct and try again.
At line:25 char:15
+ throw Exception('Input string is not a valid octet string GUID')
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Exception:String) [],

```

```
CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Evidentně máme chybu ve slově Exception. Po krátkém pátrání v helpu

```
PS C:\> help throw
TOPIC
    about_Throw
```

zjistíme, že syntaxe je jiná oproti té, uvedené ve skriptu:

```
throw Exception('Input string is not a valid octet string GUID')
```

vs.

THROWING A STRING

The optional expression in a Throw statement can be a string, as shown in

the following example:

```
C:\PS> throw "This is an error."
```

```
This is an error.
```

```
At line:1 char:6
```

```
+ throw <<<< "This is an error."
```

```
      + CategoryInfo          : OperationStopped: (This is an
error.:String) [], RuntimeException
```

```
      + FullyQualifiedErrorId : This is an error.
```

Upravíme proto mírně náš skript a spustíme jej znovu

```
else
{
    throw 'Input string is not a valid octet string GUID'
}
```

```
PS C:\> Convert-OctetStringToGuid d05db2e2-42de-4998-80cc-
90fdc8111c54
```

```
Input string is not a valid octet string GUID
```

```
At line:25 char:9
```

```
+ throw 'Input string is not a valid octet string GUID'
```

```
+ ~~~~~
```

```
      + CategoryInfo          : OperationStopped: (Input string is...tet string
GUID:String) [], RuntimeException
```

```
      + FullyQualifiedErrorId : Input string is not a valid octet
string GUID
```

Nyní již vidíme, že se nám správně zobrazila chyba. Proč jsme ale zachytili výjimku, když máme GUID ve správném tvaru? Odpověď nalezneme na tomto řádku:

```
if(32 -eq $guid.Length)
```

Délka vstupního řetězce musí být 32 znaků. Náš řetězec má ovšem znaků 36, včetně pomlček. K této části kódu bych měl dvě výhrady:

1. Čistě pocitově mi přijde logičtější postavit porovnání následujícím způsobem:
`if($guid.Length -eq 32)`
2. Délku porovnávat buď přes celý řetězec (36 znaků) a v těle skriptu je rozdělit na jednotlivé části nebo dát alespoň krátkou zmínku do nápovědy.

Zkusme tedy odstranit pomlčky a spustit funkci znovu:

```
PS C:\> Convert-OctetStringToGuid d05db2e242de499880cc90fdc8111c54  
e2b25dd0-de42-9849-80cc-90fdc8111c54
```

Voilà. Vidíte nekonzistenci mezi vstupem a výstupem (zde již s pomlčkami). Jak se vypořádat se vstupem? Máme několik možností jednou z nich je následující konstrukce, kterou můžeme umístit na začátek skriptu:

```
$Guid = $Guid -replace '-', ''
```

Tím odstraníme ze vstupního textu všechny pomlčky a dále pokračujeme originálním skriptem.

```
PS C:\> Convert-OctetStringToGuid d05db2e2-42de-4998-80cc-  
90fdc8111c54  
e2b25dd0-de42-9849-80cc-90fdc8111c54
```

Možná by bylo elegantnější vytvořit funkci, která bude pracovat v rouři a umožní nám elegantní převod více GUIDů najednou. Další úpravy, které bychom mohli použít, jsou například:

1. Odstranit středník za ukončením bloku parametrů. Jedná se o kosmetickou změnu, ale i na ní je vidět, že autor byl zřejmě programátor.
2. Dopsal bych alespoň krátkou nápovědu.
3. Kdybych chtěl mít opravdu kontrolu nad parametry, přidal bych konstrukci **[CmdletBinding()]** a parametr uvedl v konstrukci **[Parameter(...)]**
4. Možná bych uvažoval o pomocné funkci, která by nahradila konstrukci: `[Convert]::ToUInt16($guid.Substring(16, 2), 16) -as [byte]`
5. Výslednou hodnotu bych nevracel pomocí klíčového slova **return**, ale pouze jako **\$g.Guid**.

Je vidět, že i když si stáhnete řešení z internetu, měli byste umět se skripty a funkcemi pracovat. Možná to 1000x vyjde a jednou se stane něco, co jste nepředpokládali. V jednodušším případě se dostanete do právě popsané situace, v tom složitějším budete možná muset něco obnovovat ze zálohy.

Chtěl jsem dnes ukázat opět něco ze života. Vždy byste měli postupovat opatrně při přejímání cizích kódů. Máte nějaký příklad z vlastní praxe, kdy se skript choval jinak, než jste očekávali? Reakce uvítám v komentářích.

Závěrem humorná vsuvka. Víte, jak se baví PowerShell MVP? Malá ukázka toho, jak můžete pojmenovat proměnnou v PowerShellu je na GitHubu: <https://gist.github.com/klumsy/11266285> V konzoli vypadá jméno proměnné lépe než na webu – vyzkoušejte si to J

Další ukázka je poněkud zajímavější:

```
PS C:\> ${ } = 1; ${ } = 2; ${ } = 3; ${ } = 4; ${ } = 5
PS C:\> Get-Variable -Name ' *'
Name Value
----
1
2
3
4
5
```

No, občas může mít člověk těžkou hlavu i z maličkostí

SCRIPT BROWSER & SCRIPT ANALYZER

Pokud se pohybujete ve světě skriptování delší dobu, určitě jste prošli následujícím kolečkem:

1. Otevřu oblíbený prohlížeč a oblíbený vyhledávač.
2. Zadám text, pomocí kterého vyhledám skript.
3. Skript stáhnu, upravím, použiji.

Microsoft před nějakou dobou vytvořil nástroj Script Explorer, který umožňoval vyhledávat skripty z různých zdrojů a poté je vkládat přímo do ISE. Vývoj tohoto nástroje byl posléze bohužel ukončen. Nicméně požadavek komunity na podobný nástroj zde stále existoval.

V listopadu 2013 na MVP Summitu ukázal vývojový tým zárodek budoucího nového nástroje, který nazvali Script Browser. Posbírali množství komentářů a v dubnu tohoto roku vydali první oficiální verzi. Chvilí po vydání této verze byly uvolněny další dvě, které opravovaly některé chyby a zároveň zapracovaly další požadavky uživatelů. V současné době je tedy aktuální [Script Browser 1.2](#). Součástí instalace je i Script Analyzer. Co tyto nástroje umí?

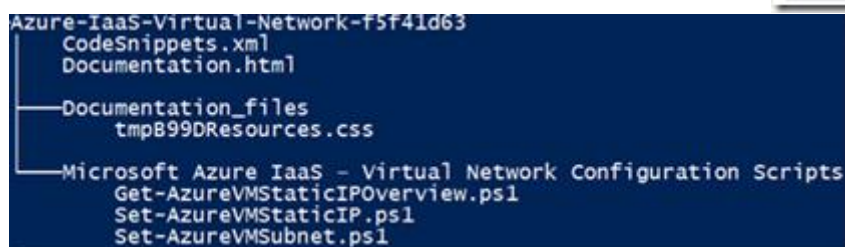
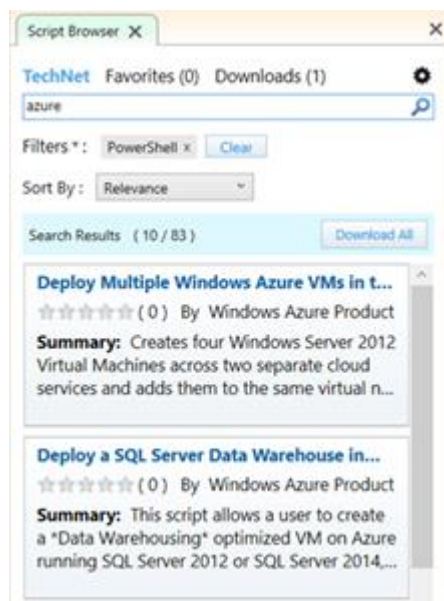
Vyhledávání skriptů

Hlavním úkolem Script Browseru je vyhledávání skriptů na TechNetu (ano, zatím pouze na https://msdnshared.blob.core.windows.net/media/TNBlogFS/prod.evol.blogs.technet.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/00/48/74/metablogapi/image_1B1E3E52.png TechNetu). Po instalaci doporučuji podívat se na Nastavení (dostupné přes ikonu ozubeného kola v pravém horním rohu). Zde si nastavte cestu k ukládání stažených skriptů.

Můžete si nastavit filtr pro vyhledávání (klikněte přímo na slovo Filters). Poté již jenom zadejte vyhledávané slovo nebo frázi. Na následujícím obrázku je vidět část výsledku při vyhledávání skriptů pro Azure.

Po nalezení skriptu je otevřete (dvojklikem či tlačítkem Open) a skript si můžete prohlédnout. Lze také jeho část (nebo samozřejmě celý) zkopírovat do schránky. Pokud chcete, můžete si skript přidat do položky Favorites.

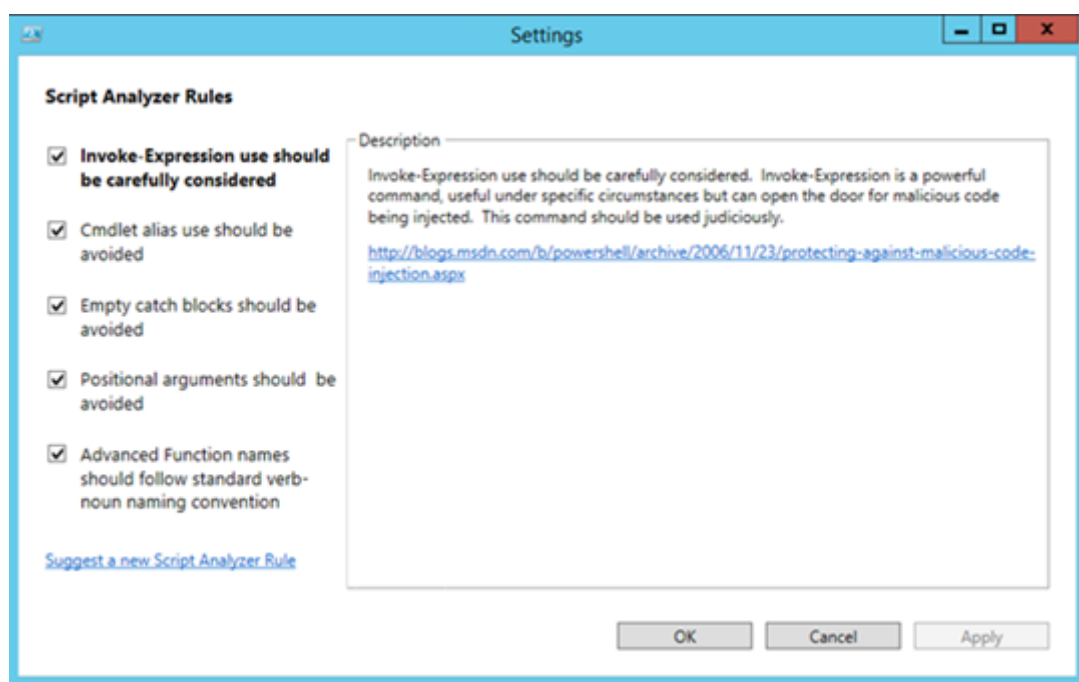
Nejzajímavější možností je samozřejmě stažení skriptu na lokální počítač. Po stažení se přidá skript na záložku Downloads a ve složce, kterou jste nastavili, se uloží jak samotné skripty, tak podpůrné soubory – nejzajímavější je HTML soubor s nápovědou, resp. kopií TechNet stránky.



Kontrola skriptů

Druhým nástrojem, který je součástí instalace je Script Analyzer. Tento nástroj vám na základě určitých pravidel doporučí změny ve vašem kódu. Pojd'me si jej ukázat na několika skriptech z TechNetu.

Nejprve se podíváme na nastavení. V rámci nastavení určíme, které části kódu bude Script Analyzer kontrolovat.



Když kliknete na určité pravidlo, zobrazí se k němu krátký popis toho, co hlídá. Například právě označené pravidlo varuje před použitím cmdletu Invoke-Expression. Pokud nechám zapnutá všechna pravidla a pustím scan skriptu, zobrazí se mi následující hlášení:

Line	Issue	Problem Element
22	cmdlet using positional parameter fo...	Write-Output "Connection entered: \$MyConnection does not exist in the automa...
40	cmdlet using positional parameter fo...	Write-Output "Certificate entered: \$MyCert does not exist in the automation serv...

Jedná se o pravidlo hlídající použití pozičních parametrů u cmdletů:

Description

Readability and clarity should be the goal of any script we expect to maintain over time. When calling a command that takes parameters, where possible consider using Named parameters as opposed to Positional parameters.

Consider the following command, calling an Azure Powershell cmdlet with 3 Positional parameters:

Set-AzureAclConfig "10.0.0.0/8" 100 "MySiteConfig" -AddRule -ACL \$AclObject -Action Permit

If the reader of this command was not familiar with the set-AzureAclConfig cmdlet, they may not know what the first 3 parameters are.

The same command called using Named parameters which is easier to read:

Set-AzureAclConfig -RemoteSubnet "10.0.0.0/8" -Order 100 -Description "MySiteConfig" -AddRule -ACL \$AclObject -Action Permit

Additional reading:

<http://blogs.technet.com/b/heyscriptingguy/archive/2012/04/22/the-problem-with-powershell-positional-parameters.aspx>

Pokud se vám některé pravidlo nelíbí a nechcete jej mít zapnuté, můžete jej jednoduše odškrtnout a ve skriptu nebude kontrolováno.

Na dalším příkladu vidíte kontrolu pravidla pro vytváření jmen funkcí. Jak víte, cmdlety by měly dodržovat jmennou konvenci Verb-Noun. Toto pravidla je dobré v rámci dobrých mravů dodržovat i pro vaše vlastní funkce:

Line	Issue	Problem Element
11	Function Name Not Following Verb-Noun Format	OtestujNeco[] Does not follow Verb-Noun format.

V plánu je vytváření dalších pravidel. I vy můžete navrhnout pravidlo, které považujete za vhodné. Pomocí linku v nastavení o něj můžete požádat zde: <http://scriptanalyzer.codeplex.com/workitem/list/basic>

Osobně mám vypnuté pravidlo pro hlídání Invoke-Expression. Pokud existuje cmdlet a já jej použiji, jsem si jist, co dělá. Stejným způsobem bych vypl navrhované pravidlo o oznamování přítomnosti cmdletu Write-Host. V některých případech mi nenávidí vůči tomuto cmdletu přijde až přehnaná J

Musím říct, že tyto dva nástroje považuji za velmi užitečné a doufám, že budou dále rozvíjeny tak, aby některé z navrhovaných funkcí byly implementovány co nejdříve. Myslím, že by si to PowerShell komunita zasloužila.

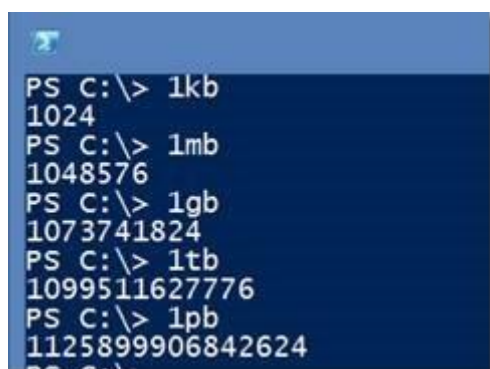
Určitě je vyzkoušejte, ještě jednou připomínám, že aktuálně poslední verze je 1.2.

KRÁTKÉ TIPY NA PRÁZDNINY

Dnes se podíváme na dva krátké tipy, které se vám mohou hodit v každodenní praxi. Upřímně řečeno – až do minulého týdne jsem netušil, jak může být první z nich tolik neznámý. Ale po jeho použití jsem uvedl v úžas několik opravdu chytrých lidí a proto se o něj s vámi také podělím. Předpokládám, že skalní fandové PowerShellu jej znají. Pokud mezi ně patříte, zkuste tento tip poslat dál. Třeba budete překvapeni, kolik lidí jej nezná.

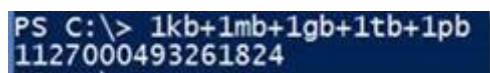
Převod mezi megy a kiloby

Určitě to znáte – čas od času je potřeba převádět v IT světě mezi kB, MB, GB případně dále. Nejhorším způsobem výpočtu je čisté násobení/dělení tisícem. To už snad dnes nikdo nedělá a většina lidí dělí s konstantou 1024. Já při každém takovém počítání startuji PowerShell a používám jeho interní jednotky. Již od v1 je možné počítat s kB, MB a GB. V dalších verzích poté byly přidány ještě TB a PB.



```
PS C:\> 1kb
1024
PS C:\> 1mb
1048576
PS C:\> 1gb
1073741824
PS C:\> 1tb
1099511627776
PS C:\> 1pb
1125899906842624
```

Oblíbený příklad Jeffreyho Snovera (tvůrce PowerShellu) je na různých prezentacích toto:



```
PS C:\> 1kb+1mb+1gb+1tb+1pb
1127000493261824
```

Standardními způsoby by vám výpočet trval o trochu déle.

Před pár dní jsem byl svědkem hloučku IT lidí, kteří se různě dohadovali a smáli nějaké (určitě vtipné) aktivitě. Když jsem přišel blíž, zjistil jsem, že přepočítávají kB na MB. Každý měl jinou metodu, ale otevřená kalkulačka na webu byla jasným znamením, kam celé snažení směřuje. Zadáním bylo zjistit, kolik je 950000kB. Trochu nesměle jsem řekl, že jim tedy ukážu, jak bych postupoval já. Myslím, že ode mě nemohli čekat nic jiného, než že řeknu: „Spust’ PowerShell.“ Po mém diktování zadal jeden z nich do okna formulku, a když se ukázal výsledek, byl jsem sám překvapen, jak všichni zareagovali. Žádný z nich tyto konstanty v PowerShellu neznal. Musím říct, že takhle jednoduchým trikem jsem již dlouho nikoho „nedostal“



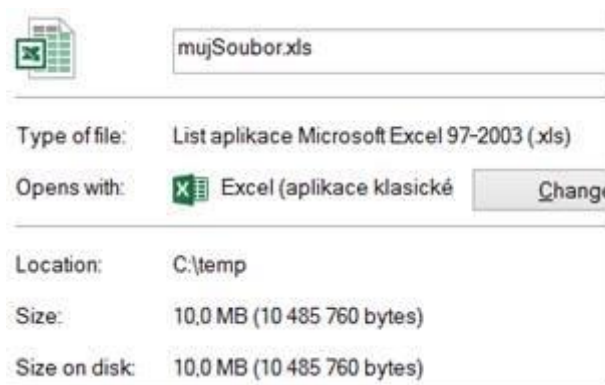
```
PS C:\> 950000kb/1mb
927,734375
PS C:\>
```

Bez nějakých složitostí mátu tu nejlepší IT kalkulačku na každém svém serveru. Jedno z dalších častých použití je např. při vytváření souborů dané velikosti. Určitě znáte fsutil a jeho použití. Pokud chcete vytvořit soubor velikosti přesně 10MB, není nic jednoduššího než:

```
PS C:\temp> fsutil file createnew mujSoubor.xls (10mb)
File C:\temp\mujSoubor.xls is created
PS C:\temp> ls .\mujSoubor.xls | ft Name, Length -auto

Name                Length
----                -
mujSoubor.xls       10485760
```

A velikost potvrdíme i přes GUI:



OGV

Dnes je moderní všude používat zkratky (já jim říkám aliasy) a jeden z nejčastějších je pro mě OGV - Out-GridView.

V poslední verzi můžete určit výstupní mód OGV a poté máte jednoduché grafické prostředí, ze kterého vám uživatelé mohou zadávat data. Naposledy jsem se s tím potkal při vytváření virtuálních počítačů ve Windows Azure. Pokud počítač vytváříte, je jedním z povinných parametrů i jméno image, ze kterého se počítač vytváří. Při standardním spuštění cmdletu **Get-AzureVMImage** vám obrazovkou projede množství informací tohoto typu:

```
Windows PowerShell

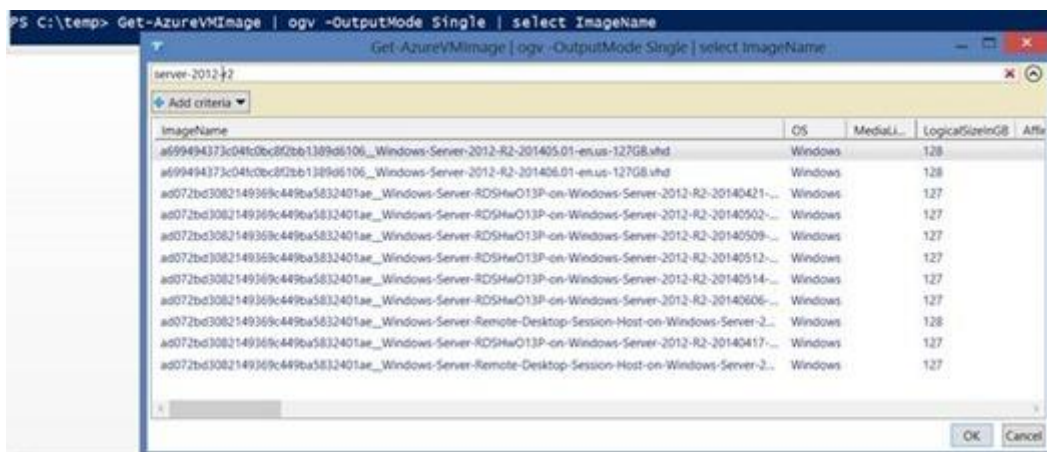
PrivacyUri      : http://go.microsoft.com/fwlink/?LinkID=287091
RecommendedVMSize : Small
PublisherName   : Microsoft Dynamics NAV Group
OperationDescription : Get-AzureVMImage
OperationId      : ed232b36-5d4a-b713-ab17-eb678ab6dd48
OperationStatus  : Succeeded

ImageName       : 9a03679de0e64e0e94fb8d7fd3c72ff1__Dynamics-NAV-2013R2-w1-RTM-201405NB.01-127GB
OS              : Windows
MediaLink       :
LogicalSizeInGB : 127
AffinityGroup   :
Category        : MSDN
Location        : East Asia;Brazil South;North Europe;West Europe;Japan East;Japan West;East US;North Central US;South Central US;West US
Label           : Microsoft Dynamics NAV 2013 R2 on Windows Server 2012
Description     : Microsoft Dynamics NAV is a business solution from Microsoft which is quick to implement, simple to use and with the power to support your business ambitions. Microsoft Dynamics NAV gives small and midsize businesses like yours complete control over your core business processes, including finance, manufacturing, warehouse management, shipping, project management, sales, services, and more.
Eula            : http://go.microsoft.com/fwlink/?LinkID=389409
ImageFamily     : Microsoft Dynamics NAV 2013 R2 on Windows Server 2012
PublishedDate   : 1. 5. 2014 2:00:00
IsPremium       : False
IconUri         : DynamicsNAV2013R2_45.png
```

Aktuálně je dostupných 228 imagů a pro vás je důležitý parametr **ImageName**. Proto v takovýchto případech rád přesměrovávám výstup do OGV a určuji výstupní mód. Do příkazové řádky zadám

```
Get-AzureVMImage | ogv -OutputMode Single | select ImageName
```

A v Out-GridView si vyfiltruji potřebný image.



Když poté výsledek potvrdím tlačítkem OK, zpátky do konzole se vrátí požadované jméno

```
PS C:\temp> Get-AzureVMImage | ogv -OutputMode Single | select ImageName
ImageName
-----
a699494373c04fc0bc8f2bb1389d6106__Windows-Server-2012-R2-201405.01-en.us-127GB.vhd
```

Tímto způsobem můžete relativně jednoduše filtrovat z velké množiny dat pouze ta, která vás zajímají.

Pokud máte nějaký vlastní tip, budu rád když se o něj podělíte v komentářích. Přeji hodně zábavy při používání PowerShellu.

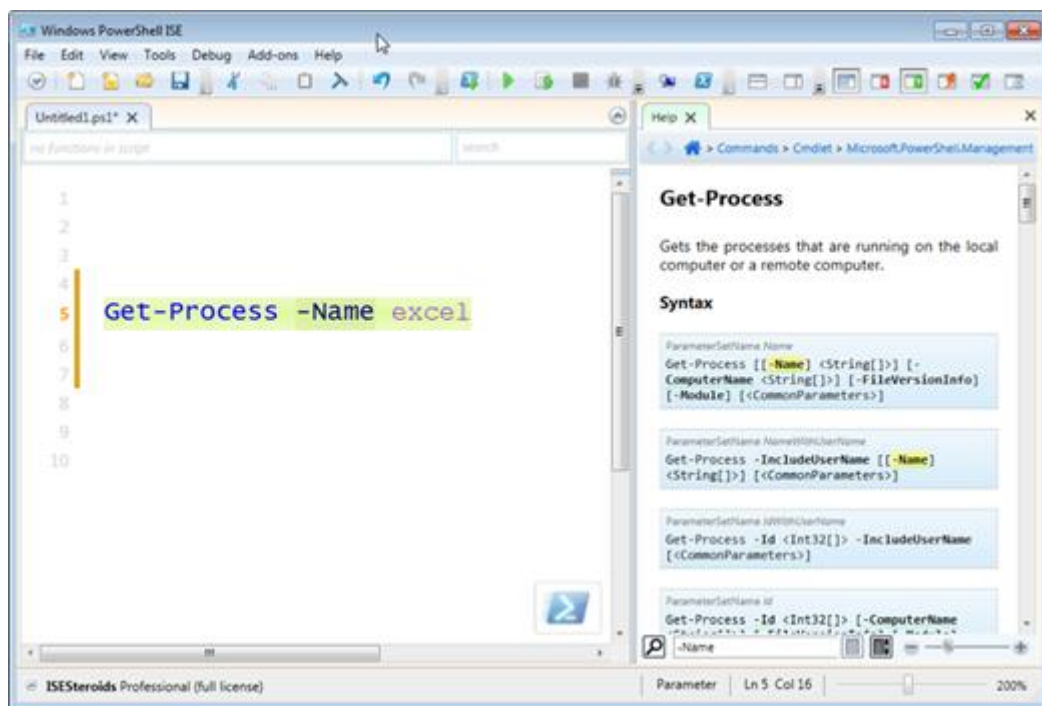
ISESTEROIDS 2.0

Před několika měsíci jsem v jednom článku zmiňoval moc povedený Add-in do PowerShell ISE – ISESteroids. Jak sám autor – Tobias Weltner – říká, jedná se o „ISE na steroidech“ – odtud i název. Já jsem měl možnost ISESteroids od první veřejné verze používat a momentálně se jedná o jediný Add-in, který startuji rovnou v ISE profilu.

Již v první verzi se jednalo o velice schopného pomocníka. Proto jsem byl zvědav na novinky, které Tobias přidal do druhé verze. Myslel jsem si, že se bude jednat o kosmetické změny, ale ISESteroids se posunuly opravdu o velký kus dopředu. Rád bych vám ukázal vlastnosti, které se mě osobně líbí nejvíc. Dopředu upozorňuji, že některé části nemají zatím jméno (nebo jej zatím neznám) a proto je budu nazývat tak, jak mi to přijde nejpřirozenější.

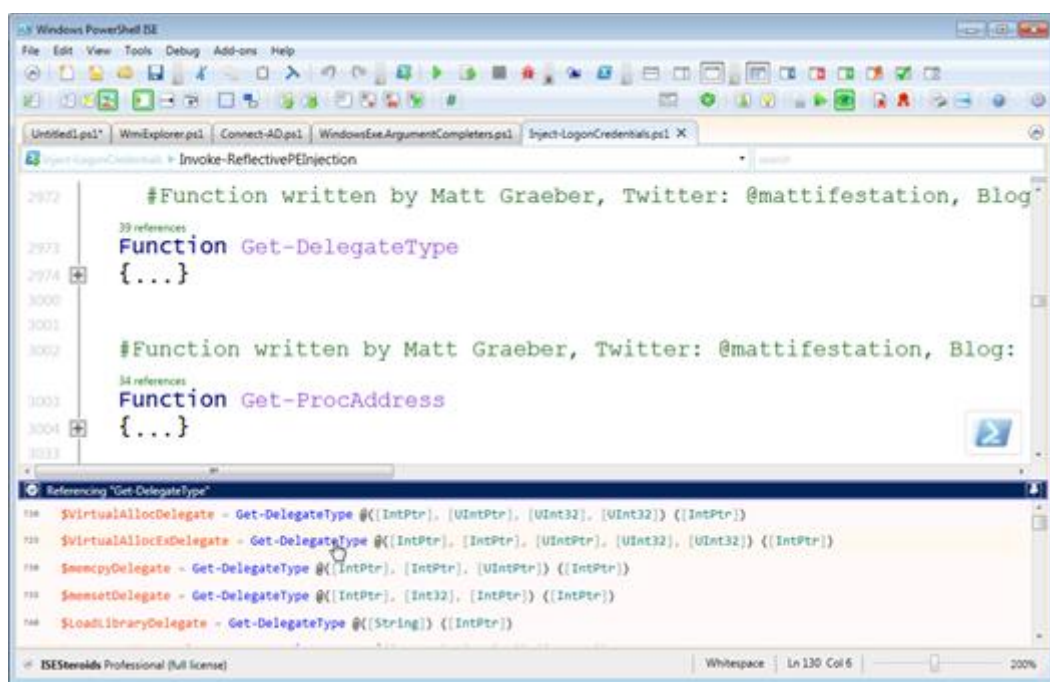
CopyFrom-Help – kopírování příkladů z nápovědy do ISE

Již v první verzi byla možnost zobrazit si v okně nápovědu, která byla zároveň lépe formátována. V nové verzi stačí kliknout na příklady v sekci Examples a kód se rovnou vloží na pozici kurzoru.



Show-Reference – odkazy na použití daného kódu

Pokud pracujete s Visual Studiem, tuto funkcionalitu již znáte. U definice funkce se objeví text s číslem, které udává, kolikrát je daná funkce odkazována. Zároveň se při kliknutí na tuto referenci ukáže okno, které zobrazuje řádky, kde je funkce použita. Jednoduchým způsobem se tedy od definice funkce můžete dostat na všechny výskyty použití.



Stejně jako v předchozí verzi existuje v ISESteroids možnost přejít na definici funkce a můžete se tedy pohybovat oběma směry (od definice k použití, od použití k definici).

Invoke-Versioning + Show-Diff – verzování dokumentu přímo z ISE

Již několik let se snažím mít přehled ve svých skriptech a jejich různých verzích. Momentálně používám git, ale uznávám, že jeho použití v příkazové řádce se nemusí líbit všem. Zatím nejbližší

mému administrátorskému přístupu je program VersionRecall od Sapienu. ISESteroids přichází s možností ukládat verze skriptu při každém uložení.

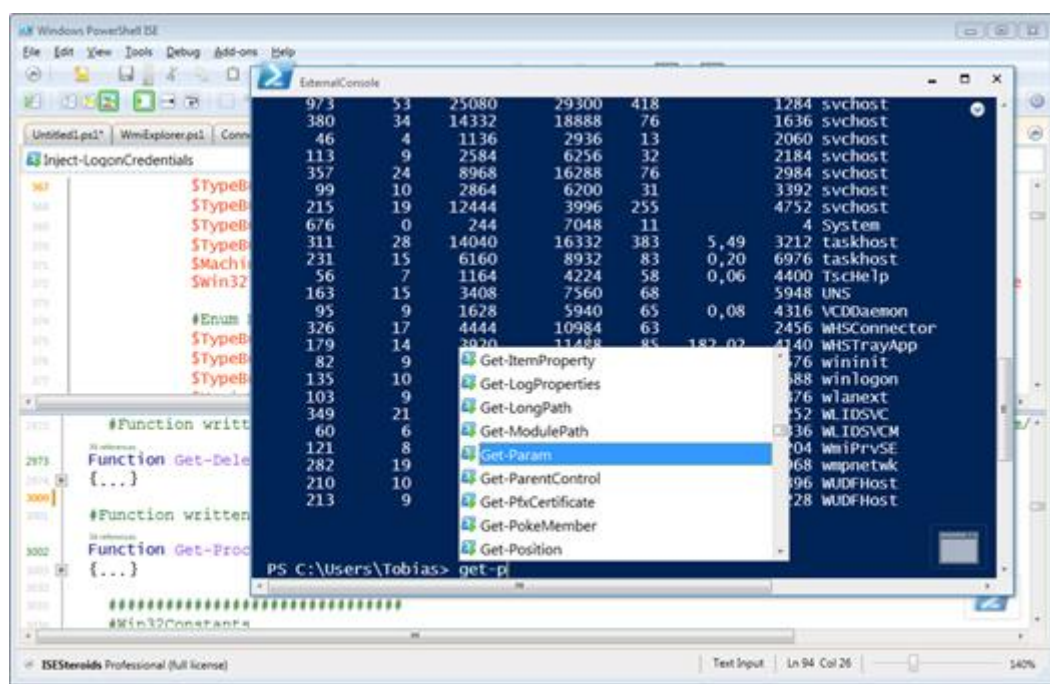
Ve speciálním okně vidíte jednotlivé verze a pokud chcete zobrazit rozdíly mezi současnou a minulou, otevře se program WinDiff, který vám rozdíly přehledně zobrazí.

Dle mého se jedná o ideální kombinaci, kdy administrátora nezajímá TFS či jiný verzovací systém, ale prostě jen píše skripty. Nestará se o přidávání do repository, commit, či jiné „vývojářské“ záležitosti. Pokud je potřeba se vrátit ke starší verzi, prostě si jen vybere potřebnou ze seznamu a vloží ji do aktuálního okna.

Osobně jsem zvědav, jak moc se tato funkcionality bude líbit, protože si myslím že v IT Pro světě je verzování obecně hodně podceňovanou záležitostí.

Invoke-ExternalConsole – ISE konzole jako undocked okno

Zajímavá vychytávka, kdy můžete konzoli v ISE odpojit „undock“ a mít ji jako samostatné okno. Po použití ji zase připnete zpátky na původní místo.

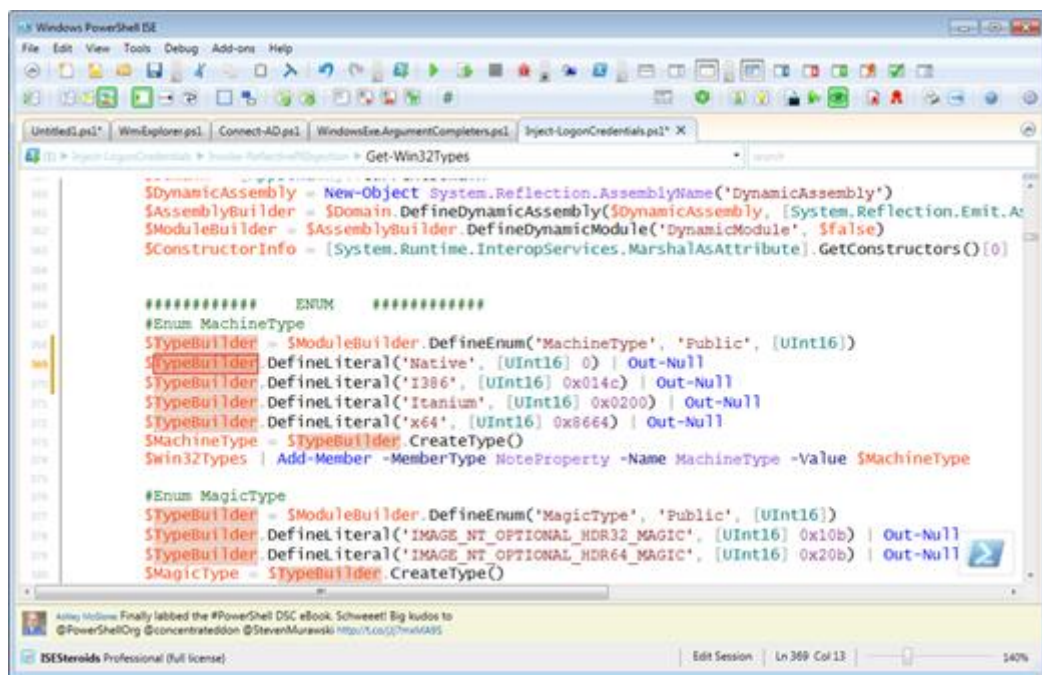


Invoke-Fix – spuštění „oprav“ kódu

Pokud píšete skripty, stane se vám, že máte svůj vlastní styl. Pokud dále skripty sdílíte chcete (doufám), aby byl skript čitelný i pro ostatní. K tomu vám může pomoci tato část ISESteroids. Na daný skript spustíte tyto „fixy“ a ony vám kód u(o)praví. Typickým příkladem je odstranění aliasů, pozičních parametrů, ...

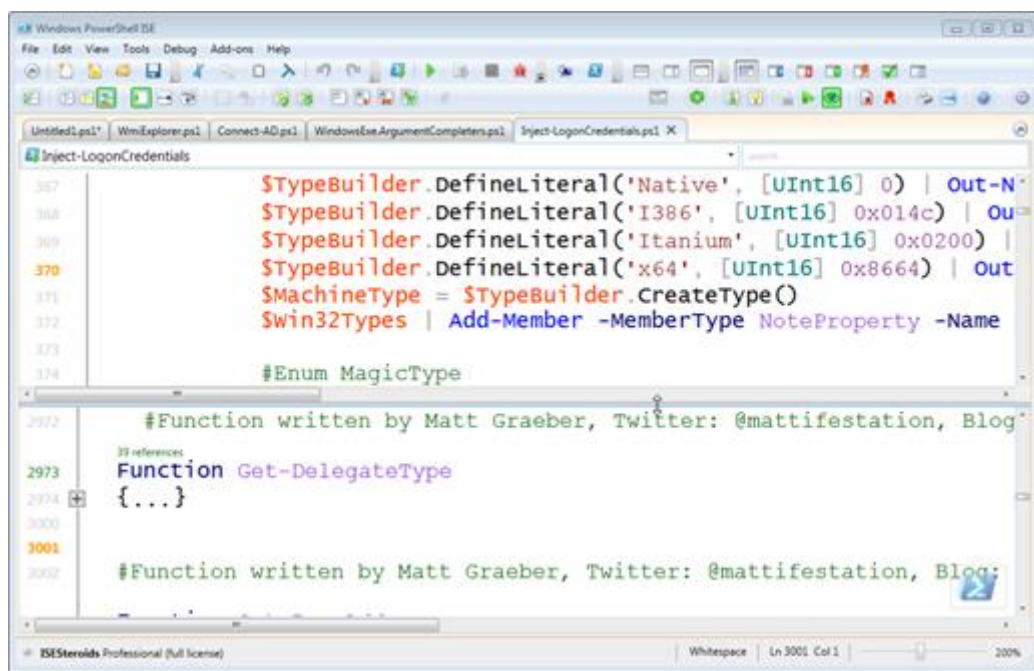
Select-MultiText – označení stejného textu napříč dokumentem

Aneb rychlé najdi/nahrad'. V ISE je od verze 3 možnost použití sloupcových bloků. ISESteroids tuto myšlenku přivádí ještě o krok dále. Můžete označit určité slovo (např. proměnnou) a jednoduše ji přepsat v celém dokumentu.



Split View – rozdělení okna

Tuto funkcionalitu také znáte z jiných programů. V případě, že se chcete podívat na různá místa jednoho dokumentu, můžete použít vodorovné rozdělení oken.



Oficiální uvedení

ISESteroids v2 jsou zatím v neveřejném testování. Oficiální představení proběhne v rámci evropského PowerShell Summitu v Amsterdamu. Pokud se vám ISESteroids líbí, doporučuji ještě ke shlédnutí oficiální video na stránkách <http://www.powertheshell.com/isesteroids2/> (2014)

VZDÁLENÝ PŘÍSTUP POPRVÉ

Tímto článkem bych chtěl zahájit sérii popisující nastavování a správu vzdáleného přístupu v PowerShelli. Mým cílem bude provést váš všemi možnými scénáři a případnými pastmi.

Nejprve (dnes) se podíváme na ideální situaci, kdy nám pro nastavení stačí opravdu minimum znalostí (spustit PowerShell). Příště si sjednotíme názvosloví, abych se v dalších dílech mohl opět věnovat praktickým ukázkám. Postupně se podíváme na vytváření vlastních endpointů, práci mimo doménu, možnosti konfigurace vzdáleného připojení tak, aby uživatelé mohli provádět pouze určité akce a celou sérii bych rád zakončil ukázkami řešení problémů a nastavením PowerShell Web Access.

Ideální situace – doména

Většina nás bude pracovat v doméně, kde je také nastavení vzdáleného přístupu nejjednodušší. Proto si nastavení ukážeme na tomto příkladu jako první. Máme jeden server jako DC s Windows 2012 Server OS (MTSHOWVMDC) a dva klientské počítače: Windows 7 (MTSHOWVMW7) a Windows 8 (MTSHOWVMW8).

Na DC je remote access nastaven standardně a není tedy potřeba nic nastavovat. Zkusíme se připojit z Win 7 stanice.

```
PS C:\> Enter-PSSession mtshowvmc
[mtshowvmc]: PS C:\Users\moravec\Documents>
```

Enter-PSSession se používá pro připojení ke vzdálenému počítači (takový PowerShell telnet). Můžete použít i alias **etsn**. V okamžiku, kdy jsme připojeni, se změní příkazový řádek a jako první část vidíte jméno počítače, na který jste připojeni. Nyní jsou všechny operace prováděny na vzdáleném počítači.

```
[mtshowvmc]: PS C:\Users\moravec\Documents> cd\
[mtshowvmc]: PS C:\> $env:COMPUTERNAME
MTSHOWVMDC
[mtshowvmc]: PS C:\> exit
PS C:\>
```

Vzdálené připojení zrušíte pomocí příkazu **exit**. Jak vidíte v ideálním případě je vzdálený přístup nakonfigurován a vy již nemusíte nic řešit. Pojdme se zkusit připojit na stanici s Windows 7. Připojujeme se z DC.

```
PS C:\Users\moravec> Enter-PSSession mtshowvmw7
```

```
Enter-PSSession : Connecting to remote server mtshowvmw7 failed with the
following error message : The client cannot connect to the destination
specified in the request. Verify that the service on the destination is
running and is accepting requests. Consult the logs and documentation for the
WS-Management service running on the destination, most commonly IIS or WinRM.
If the destination is the WinRM service, run the following command on the
destination to analyze and configure the WinRM service: "winrm quickconfig".
For more information, see the about_Remote_Troubleshooting Help topic.
```

```
At line:1 char:1
```

```
+ etsn mtshowvmw7
```

```
+ ~~~~~
```

```
+ CategoryInfo          : InvalidArgument: (mtshowvmw7:String) [Enter-PSSession],
PSRemotingTransportException
```

```
+ FullyQualifiedErrorId : CreateRemoteRunspaceFailed
```

Spojení se nepodařilo a v chybovém hlášení je navržen i postup, jak ověřit (zapnout) vzdálený přístup. Přepneme se zpět na Win7.

```
PS C:\Users\moravec> Test-WSMan
```

```
Test-WSMan : The client cannot connect to the destination specified in the
request. Verify that the service on the destination is running and is
accepting requests. Consult the logs and documentation for the WS-Management
service running on the destination, most commonly IIS or WinRM. If the
destination is the WinRM service, run the following command on the destination
```

to analyze and configure the WinRM service: "winrm quickconfig".

At line:1 char:11

+ Test-WSMan <<<<

+ CategoryInfo : InvalidOperation: (:) [Test-WSMan],
InvalidOperationException

+ FullyQualifiedErrorId :
WsManError,Microsoft.WSMan.Management.TestWSManCommand

Co přesně **Test-WSMan** znamená, si ukážeme v některém z následujících dílů. Nyní vemte na vědomí, že to je jeden z cmdletů, kterým můžete zkontrolovat, že vám na lokálním počítači bude vzdálený přístup fungovat. Doporučení je opět stejné – spustit „**winrm quickconfig**“. Pokud budete pátrat, přijdete na to, že *winrm* je VBS soubor (více než 4000 řádek), který nastaví vzdálený přístup. Nevím jak vy, ale já si svou PowerShell konzoli nechci VBS skriptem zašpinit. Proto radši používám cmdlet **Enable-PSRemoting**. Spustím jej proto z PowerShellu na počítači Win7.

PS C:\Users\moravec> Enable-PSRemoting

Enable-PSRemoting : Access is denied. To run this cmdlet, start Windows PowerShell with the "Run as administrator" option.

At line:1 char:1

+ Enable-PSRemoting

+ ~~~~~

+ CategoryInfo : NotSpecified: (:) [Enable-PSRemoting],
InvalidOperationException

+ FullyQualifiedErrorId :
System.InvalidOperationException,Microsoft.PowerShell.Commands.EnablePSRemotingCommand

Ah – vzhledem k tomu, že budeme ovlivňovat nastavení systému, je potřeba konzoli spustit jako administrátor. Druhý pokus již bude úspěšnější.

PS C:\> Enable-PSRemoting

WinRM Quick Configuration

Running command "Set-WSManQuickConfig" to enable this machine for remote management through WinRM service.

This includes:

1. Starting or restarting (if already started) the WinRM service
2. Setting the WinRM service type to auto start
3. Creating a listener to accept requests on any IP address
4. Enabling firewall exception for WS-Management traffic (for http only).

Do you want to continue?

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

WinRM has been updated to receive requests.

WinRM service type changed successfully.

WinRM service started.

WinRM has been updated for remote management.

Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.

WinRM firewall exception enabled.

Confirm

Are you sure you want to perform this action?

Performing operation "Registering session configuration" on Target "Session configuration "Microsoft.PowerShell32" is not found. Running command

"Register-PSSessionConfiguration Microsoft.PowerShell32 -processorarchitecture x86 -force" to create "Microsoft.PowerShell32" session configuration. This

```
will restart WinRM service.".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is
"Y"):
PS C:\>
```

Jak vidíte, ve skutečnosti se provede několik akcí souvisejících s nastavením vzdáleného přístupu. Které to jsou a v jakém pořadí je vidět ve výpisu.

Nyní můžeme provést opět kontrolu:

```
PS C:\Users\moravec> Test-WSMan

wsid : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 2.0
```

A můžeme se připojit z DC, tentokrát úspěšně.

```
PS C:\Users\moravec> Enter-PSSession mtshowvmw7
[mtshowvmw7]: PS C:\Users\moravec\Documents>
```

Při všech současných pokusech o připojení jsem používal účet doménového administrátora (moravec). Ne vždy je to ale chtěný stav. Většinou máte operátory, kterým chcete připojení umožnit, ale nechcete, aby zároveň patřili do skupiny Domain Admins. Na závěr si tedy dnes ukážeme, jak nastavit přístup pro neadminy.

Nastavení práv pro vzdálený přístup

Máme vytvořený účet psrausr, který je členem pouze skupiny Domain Users. Zkusíme se s ním připojit na vzdálený počítač.

```
PS C:\> $env:username
psrausr
PS C:\> Enter-PSSession mtshowvmw7
```

```
Enter-PSSession : Connecting to remote server mtshowvmw7 failed with the
following error message : Access is denied. For more information, see the
about_Remote_Troubleshooting Help topic.
```

```
At line:1 char:1
```

```
+ Enter-PSSession mtshowvmw7
```

```
+ ~~~~~
```

```
+ CategoryInfo          : InvalidArgument: (mtshowvmw7:String) [Enter-PSSession],
PSRemotingTransportException
```

```
+ FullyQualifiedErrorId : CreateRemoteRunspaceFailed
```

Nyní na cílovém počítači přidáme tento účet do skupiny administrators.

```
PS C:\> net localgroup administrators showdom\psrausr /add
The command completed successfully.
```

```
PS C:\> net localgroup administrators
Alias name administrators
Comment Administrators have complete and unrestricted access to the
computer/domain
```

Members

```
-----
-
admin
Administrator
SHOWDOM\Domain Admins
```

```
SHOWDOM\psrausr
The command completed successfully.
```

A nyní se již ze vzdáleného počítače připojíme.

```
PS C:\> Enter-PSSession mtshowvmw7
[mtshowvmw7]: PS C:\Users\psrausr\Documents> $env:username
psrausr
[mtshowvmw7]: PS C:\Users\psrausr\Documents> $env:computername
MTSHOWVMW7
[mtshowvmw7]: PS C:\Users\psrausr\Documents> exit
```

Vzhledem k tomu, že skupina administrators má standardně povolen přístup, tato metoda funguje. Odebereme účet:

```
PS C:\> net localgroup administrators showdom\psrausr /delete
The command completed successfully.
```

A podíváme se, jak zjistit, kdo se může vlastně na náš počítač připojit. Trošku přeskočím do dalších lekcí a prozradím, že pokud se připojujete na vzdálený počítač, připojujete se na jeden z předdefinovaných vstupních bodů. Jaká je konfigurace toho, který jsme využívali v předchozích příkladech?

```
PS C:\> Get-PSSessionConfiguration -Name microsoft.powershell | Format-List *
```

Name	: microsoft.powershell
Filename	: %windir%\system32\pwrshplugin.dll
SDKVersion	: 1
XmlRenderingType	: text
lang	: en-US
PSVersion	: 2.0
ResourceUri	:
	http://schemas.microsoft.com/powershell/microsoft.powershell
SupportsOptions	: true
Capability	: {Shell}
xmlns	:
	http://schemas.microsoft.com/wbem/wsman/1/config/PluginConfiguration
Uri	:
	http://schemas.microsoft.com/powershell/microsoft.powershell
ExactMatch	: true
SecurityDescriptorSddl	:
	O:NSG:BAD:P(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
Permission	: BUILTIN\Administrators AccessAllowed

Všimněte si poslední řádky. Na ní je vysvětleno, proč se administrátoři mohou připojit.

Nebojte se, pokud jste poslední ukázky nepochopili. V následujících dílech bude vše podrobně vysvětleno. Dnes bylo důležité ukázat si, že nastavení vzdáleného přístupu v doméně je jednoduché a nic vám nebrání k tomu, abyste jej na vašich počítačích zapnuli.

A ještě jedna poznámka na závěr. ISESteroids v2, o kterých jsem psal minule už je uvolněn, můžete začít zkoušet. Některé věci, které ani v pilotu nebyly, byly přidány navíc. Já jsem zatím nadšen.

ZPRACOVÁNÍ TEXTU

V dnešním textu se podíváme na zpracování textu a na použití PowerShellu při potřebě „rychlé akce“. Jsem právě v Redmodu v kampusu Microsoftu na letošním MVP Summitu. Stejně jako každá jiná konference, i tato je nabitá přednáškami od pondělí až do pátku. Oficiální program začíná již v neděli a není proto divu, že po několika dnech už tělo začíná protestovat.

Důležité je vybrat si správné prezentace. Mimo těch, které se týkají mé oblasti, je možné sáhnout do podobných oborů, což využívám v případě Microsoft Azure. I když mám již svůj program nahrán v telefonu, občas se hodí kouknout se, jestli se nezměnilo téma nějaké prezentace nebo se prostě neobjevila nějaká zajímavější.

Všechny prezentace máme přehledně na webu, takže není problém si je stáhnout do textového souboru, např. pomocí nám již známého `Invoke-WebRequest`. Nebo čistým copy-paste, jak jsem to udělal já. Vyhledávání v tomto souboru bylo ovšem trochu složitější a tak jsem otevřel konzoli a snažil se trochu si ulehčit práci.

Pozn.: Témata jednotlivých prezentací jsou bohužel NDA, proto jsem pro účely tohoto článku upravil některé názvy.

Pozn. 2: Na dnešním příkladě bych chtěl ukázat, že pro rychlé akce náhodného typu můžete PowerShell využít k klidným svědomím. Celé následně popsání řešení bylo otázkou zhruba pěti minut při přejezdu mezi budovami, kde se přednášky konaly.

Nejprve jsem si čistě vylistoval obsah souboru pomocí `Get-Content` a následně zjistil, že v Notepadu je hledání samozřejmě pohodlnější. Nicméně jsem přešel na těžší váhu: `Select-String`. Nejprve se podíváme na řádky s aktuálním dnem:

```
PS C:\> Select-String $path -Pattern thur
```

```
Users\Makovec\Documents\fri.txt:3:Thursday, November 6, 09:00 - 10:00,  
Building/Room: MS Campus 33/room1
```

```
Users\Makovec\Documents\fri.txt:13:Thursday, November 6, 09:00 - 09:30,  
Building/Room: MS Campus 33/room2
```

```
Users\Makovec\Documents\fri.txt:23:Thursday, November 6, 09:00 - 10:30,  
Building/Room: MS Campus 34/room3
```

```
Users\Makovec\Documents\fri.txt:32:Thursday, November 6, 13:30 - 15:00,  
Building/Room: MS Campus 33/room3
```

`Select-String` hledá text na základě regulárního výrazu. V našem případě to nebudeme potřebovat, ale nebudeme příkaz měnit. Pokud byste chtěli čistě textové hledání, použijte switch `SimpleMatch`.

Dostali jsme zajímavý výpis, ale moc užitečný nám není. Zkusme přidat parametr `Context`. Tímto parametrem říkáme kolik řádek před a po aktuálním nalezeném textu chceme zobrazit.

```
PS C:\> Select-String $path -Pattern thur -Context 2,0
```

```
Users\Makovec\Documents\fri.txt:1:.NET a jeho vyuziti pri uprave zahrady:  
Introduction
```

```
Users\Makovec\Documents\fri.txt:2:(AB21) 36 spots of 88 are available for this  
session.
```

```
> Users\Makovec\Documents\fri.txt:3:Thursday, November 6, 09:00 - 10:00,  
Building/Room: MS Campus/room1
```

```
Users\Makovec\Documents\fri.txt:11:Access for corps
```

```
Users\Makovec\Documents\fri.txt:12:(AB13) 111 spots of 144 are available for  
this session.
```

```
> Users\Makovec\Documents\fri.txt:13:Thursday, November 6, 09:00 - 09:30,  
Building/Room: MS Campus/room2
```

```
Users\Makovec\Documents\fri.txt:21:SQL Server 2005 - novinky
```

```
Users\Makovec\Documents\fri.txt:22:(SD12) 10 spots of 20 are available for  
this session.
```

```
> Users\Makovec\Documents\fri.txt:23:Thursday, November 6, 09:00 - 10:30,
Building/Room: MS Campus/room3
```

```
Users\Makovec\Documents\fri.txt:30:Access & Excel
```

Ha – vypadá to, že máme výsledek. Tedy, téměř. Samozřejmě že stále máme text, který nevypadá moc pěkně. Zkusíme ho trochu vylepšit.

Zde se nám hodí náš starý kamarád – *Get-Member*.

```
PS C:\> Select-String $path -Pattern thur -Context 2,0 | gm
```

```
TypeName: Microsoft.PowerShell.Commands.MatchInfo
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
RelativePath	Method	string RelativePath(string directory)
ToString	Method	string ToString(), string ToString(string directory)
Context {get;set;}	Property	Microsoft.PowerShell.Commands.MatchInfoContext Context
Filename	Property	string Filename {get;}
IgnoreCase	Property	bool IgnoreCase {get;set;}
Line	Property	string Line {get;set;}
LineNumber	Property	int LineNumber {get;set;}
Matches {get;set;}	Property	System.Text.RegularExpressions.Match[] Matches
Path	Property	string Path {get;set;}
Pattern	Property	string Pattern {get;set;}

Docela zajímavě vypadá vlastnost *Context*.

```
PS C:\> Select-String $path -Pattern thur -Context 2,0 | select context
```

```
Context
```

```
-----
```

```
Microsoft.PowerShell.Commands.MatchInfoContext
```

```
Microsoft.PowerShell.Commands.MatchInfoContext
```

```
Microsoft.PowerShell.Commands.MatchInfoContext
```

```
Microsoft.PowerShell.Commands.MatchInfoContext
```

```
Users\Makovec\Documents\fri.txt:31:(CD42) 125 spots of 144 are available for
this session.
```

```
> Users\Makovec\Documents\fri.txt:32:Thursday, November 6, 13:30 - 15:00,
Building/Room: MS Campus/room3
```

Což není úplně přesně to, co jsme chtěli. Nicméně víme, že pro podívání se dovnitř výsledného objektu pomocí parametru *ExpandProperty*.

```
C:\> Select-String $path -Pattern thur -Context 2,0 | select -expand context
PreContext          PostContext DisplayPreContext DisplayPostContext
-----
{.NET a jeho vyuziti pri u... {}          {\.NET a jeho vyuziti pri u... {}
{Access for corps, (AB13) ... {}          {Access for corps, (AB13) ... {}
{SQL Server 2005 - novinky... {}          {SQL Server 2005 - novinky... {}
{Access & Excel, (CD42) ... {}          {Access & Excel, (CD42) ... {}
```

PreContext se jeví jako zajímavá možnost.

```
PS C:\> Select-String $path -Pattern thur -Context 2,0 | select -
ExpandProperty Context | Select -Expand PreContext
```

```
.NET a jeho vyuziti pri uprave zahradky: Introduction
(AB21) 36 spots of 88 are available for this session.
Access for corps
(AB13) 111 spots of 144 are available for this session.
SQL Server 2005 - novinky
(SD12) 10 spots of 20 are available for this session.
Access & Excel
(CD42) 125 spots of 144 are available for this session.
```

Nyní již máme zhruba to, co jsme chtěli na začátku. Jenom by bylo dobré vědět, od kolika hodin se přednášky konají. Pohledem na výstup *Get-Member* zjistíme, kde by se mohla vyskytnout požadovaná informace a pomocí *Format-List*si můžeme naši teorii ověřit.

```
PS C:\> Select-String $path -Pattern thur -Context 2,0 | fl *
IgnoreCase : True
LineNumber : 3
Line : Thursday, November 6, 09:00 - 10:00, Building/Room: MS Campus/room1
Filename : fri.txt
Path : C:\Users\Makovec\Documents\fri.txt
Pattern : thur
Context : Microsoft.PowerShell.Commands.MatchInfoContext
Matches : {Thur}
```

Vidíme, že potřebné info je ve vlastnosti *Line*. Trošku si změním kód, protože budeme skládat informace z několika vlastností dohromady.

```
PS C:\> foreach($r in (Select-String $path -Pattern thur -Context 2,0)) {
$r.Context.PreContext }
```

Výsledek je pořád stejný, můžeme pokročit dále. V tomto okamžiku bych asi normálně přepl do ISE, ale vzhledem k tomu, že jsem si s sebou bral pouze můj Surface RT, kde ISE není, zůstal jsem nadále v konzoli a celé řešení nakonec tvořil dále jako one-liner.

```
PS C:\> foreach($r in (Select-String $path -Patternthur -Context 2,0)) {
"$($r.line)`n$($r.Context.PreContext[0])" }
```

Thursday, November 6, 09:00 - 10:00, Building/Room: MS Campus/room1

.NET a jeho využití při úpravě zahrady: Introduction

Thursday, November 6, 09:00 - 09:30, Building/Room: MS Campus/room2

Access for corps

Thursday, November 6, 09:00 - 10:30, Building/Room: MS Campus/room3

SQL Server 2005 - novinky

Thursday, November 6, 13:30 - 15:00, Building/Room: MS Campus/room3

Access & Excel

Nyní vidíme datum i jméno přednášky. Vzhledem k tomu, že výsledek se mi ještě tolik nelíbil, upravil jsem jej ještě trochu. Dále uvádím pouze část, která zobrazuje informace a pro přehlednost jsem ji rozdělil na jednotlivé řádky:

<code>\$(\$r.line.Substring(22,13))</code>	Z vyhledaného textu vybere část s časem prezentace
<code>\$(\$r.context.precontext[0])</code>	Jméno prezentace
<code>\$(" "*13)</code>	Na nové řádce vypíše 13x mezeru. Tuto techniku můžete využít v různých scénářích
<code>\$(\$r.context.precontext[1].Substring(0,7))</code>	Identifikační kód prezentace
<code>\$((\$r.line -split ':')[3].trim())</code>	Místo, kde se prezentace bude konat

Nyní máme všechny potřebné informace pohromadě. Proto je spojíme do jednoho příkazu.

```
PS C:\> foreach($r in (Select-String -Path $path -Pattern 'thu' -Context 2,0))
{ "$($r.line.Substring(22,13)): $($r.context.precontext[0])`n $($(" "*13)
($r.context.precontext[1].Substring(0,7)) at $($($r.line -split
':')[3].trim())`n" }
```

09:00 - 10:00: .NET a jeho využití při úpravě zahrady: Introduction

(AB21) at MS Campus/room1

09:00 - 09:30: Access for corps

(AB13) at MS Campus/room2

09:00 - 10:30: SQL Server 2005 - novinky

(SD12) at MS Campus/room3

13:30 - 15:00: Access & Excel

(CD42) at MS Campus/room3

Pro jednodušší čtení jsem si ještě příkaz upravil tak, aby mi zobrazoval informace pouze za určitý časový interval, nicméně to jsem již dodělával po zmiňované cestě autobusem a věřím, že to pro vás bude případně pěkné cvičení.

Když tak koukám na výsledek, hodilo by se, kdyby informace nebyly pouze textové, ale ve formě objektu. V návazném díle si ukážeme, jak toho dosáhnout s daleko menším úsilím pomocí cmdletu nově dostupného v PowerShellu v5.

ZPRACOVÁNÍ TEXTU, VNEXT

V [minulém díle](#) jsme se podívali na to, jak zpracovávat textové informace pomocí cmdletu Select-String. Jak jste si možná sami vyzkoušeli, jednalo se o metodu funkční, nicméně pro složitější zpracování se může jednat o metodu ne zcela jednoduchou. Vzhledem k tomu, že zpracování textu patří k častým úkolům, objevil se v novém PowerShellu (v5) nový cmdlet, který zpracování textu zjednodušuje. Není potřeba znát regulární výrazy, což je asi největší výhodou.

PowerShell v5 si můžete vyzkoušet i na vašich počítačích. Microsoft uvolňuje zhruba jednou za měsíc novou verzi, poslední je momentálně z listopadu a naleznete ji na <http://www.microsoft.com/en-us/download/details.aspx?id=44987>

Oním novým cmdletem je ConvertFrom-String. Cmdlet funguje na základě přečtení šablony. Z této šablony poté získá informaci a dodaném textu a tento text pak dokáže zpracovat. Pojdme si ukázat zpracování textu z minulého dílu. První dva záznamy jsou následující:

```
.NET a jeho vyuziti pri uprave zahrady: Introduction
(AB21) 36 spots of 88 are available for this session.
Thursday, November 6, 09:00 - 10:00, Building/Room: MS Campus/room1
Session Type: Side Session
Speaker(s): Lucian Wischik, Managed Language Team
Hosting Expertise(s): multiple identified
Invited Expertise(s): multiple identified
Invited Interest(s): none identified
Description
```

```
Access for corps
(AB13) 111 spots of 144 are available for this session.
Thursday, November 6, 09:00 - 09:30, Building/Room: MS Campus/room2
Session Type: Side Session
Speaker(s): Rob Sinclair
Hosting Expertise(s): none identified
Invited Expertise(s): multiple identified
Invited Interest(s): none identified
Description
```

Tyto dva záznamy si uložíme do souboru, který nazveme templ.txt. Tuto šablonu si nyní otevřeme např. v notepadu a vložíme do ní značky pro ConvertFrom-String.

```
{Name*:.NET a jeho vyuziti pri uprave zahrady: Introduction}
{Id:(AB21)} 36 spots of 88 are available for this session.
Thursday, November 6, {Time:09:00 - 10:00}, Building/Room: {Place:MS
Campus/room1}
Session Type: Side Session
Speaker(s): Lucian Wischik, Managed Language Team
Hosting Expertise(s): multiple identified
Invited Expertise(s): multiple identified
Invited Interest(s): none identified
Description
```

```
{Name*:Access for corps}
{Id:(AB13)} 111 spots of 144 are available for this session.
```

Thursday, November 6, {Time:09:00 - 09:30}, Building/Room: {Place:MS
Campus/room2}
Session Type: Side Session
Speaker(s): Rob Sinclair
Hosting Expertise(s): none identified
Invited Expertise(s): multiple identified
Invited Interest(s): none identified
Description

Pro jednoduchost jsem značky označil tučným písmem. Vzhledem k tomu, že soubor má stejnou strukturu, lze jednoduše najít části, které nás zajímají. Text, který chceme ze zdroje získat, ohraničíme takto: {<Jmeno>:<text>}

Ve složených závorkách nejprve uvedeme jméno, které bude posléze přiřazeno jako vlastnost výslednému objektu a původní text oddělíme dvojtečkou. Hvězdička uvádí, že záznam se vyskytuje v souboru opakovaně a v našem případě znázorňuje začátek každého záznamu. Výsledkem by tedy měl být objekt s vlastnostmi Name, Id, Time, Place.

Nyní si ukážeme, jak funguje ConvertFrom-String.

```
PS C:\Temp> ConvertFrom-String -TemplateFile .\templ.txt -InputObject (Get-Content .\fri.txt -Raw) | Select Name, Id, Time, Place
```

Name	Id	Time	Place
----	--	----	-----
.NET a jeho vyuziti pri up...	(AB21)	09:00 - 10:00	MS Campus/room1
Access for corps	(AB13)	09:00 - 09:30	MS Campus/room2
SQL Server 2005 - novinky	(SD12)	09:00 - 10:30	MS Campus/room3
Access & Excel	(CD42)	13:30 - 15:00	MS Campus/room3

Jako šablonu voláme náš připravený soubor a jako procesovaný text je obsah souboru s vlastními daty. Vstupní data by šla do cmdletu poslat i přes rouru.

Výhodou ukázaného přístupu je jednoduchost, s jakou můžete připravit vstupní šablonu bez znalosti regulárních výrazů. A pokud byste si chtěli práci ještě zjednodušit, můžete použít add-in ISESteroids, o kterém jsem již psal, a využít jeho grafického prostředí pro přípravu souboru se šablonou.

Vánoce

A jelikož jsou ty Vánoce, můžete si v PowerShellu zkusit pustit jeden letitý skript:

<http://ye110wbeard.wordpress.com/2009/12/18/powershell-%E2%80%93-oh-christmas-tree-oh-christmas-tree-%E2%80%93-one-more-change/>

No, kdo říká, že v PowerShellu musíte být vždy produktivní

POWERSHELL A AZURE REST API

Jsem si jist, že na TechNet blogu sledujete Azure články mého kolegy z Mainstream Technologies – Matouše Rokose. Na Azure můžete jít přes portál nebo (mnou preferovaná varianta) z konzole PowerShellu. V minulosti jsem o modulu pro Microsoft Azure také psal. Občas se ale stane, že Azure cmdlety nepokrývají funkcionalitu, kterou zrovna chcete řešit. V tom případě máte možnost zaprogramovat si v C# (či jiném jazyku) nebo použít pro přístup do Azure REST API. Nebojte, nejedná se o žádnou složitost. Vzhledem k tomu, že v PowerShellu existuje cmdlet Invoke-RestMethod, je použití jednoduché – minimálně pro toho, kdo dokáže v PowerShellu pracovat např. s Active Directory.

Pokud vás zajímá víc informací v Azure REST API, podívejte na MSDN

<http://msdn.microsoft.com/en-us/library/azure/ee460799.aspx>.

Příklady

Ukážeme si dva jednoduché příklady použití REST API. V prvním si zobrazíme Azure Cloud Services. Na stránce, kterou jsem uvedl výše, najdeme, že URI pro náš dotaz má formát: `https://management.core.windows.net/<subscription-id>/services/hostedservices` a musíme použít metodu GET.

Request

The **List Cloud Services** request may be specified as follows. Replace `<subscription-id>` with your subscription ID.

Method	Request URI
GET	<code>https://management.core.windows.net/<subscription-id>/services/hostedservices</code>

Každý REST dotaz musí obsahovat (minimálně) informaci o použité verzi. Doporučuji najít si aktuální poslední verzi a tu použít.

Request Headers

The following table describes the request headers.

Request Header	Description
<code>x-ms-version</code>	Required. Specifies the version of the operation to use for this request. This header should be set to 2009-10-01 or higher. For more information about versioning headers, see Service Management Versioning .
<code>x-ms-continuation-token</code>	Optional. Specifies a continuation token that enables you to get the remainder of the response when there are more cloud services to list than can be returned in the allotted time. The value of this token is returned in the header of a previous response and will only be returned if additional cloud services need to be listed.

Nyní máme pohromadě potřebné informace (cmdlet, URI a hlavičku) a můžeme zkusit sestavit náš dotaz.

```
PS C:\> $uri =  
'https://management.core.windows.net/9a89...0b44/services/hostedservices'  
PS C:\> Invoke-RestMethod -Uri $uri
```

```
Invoke-RestMethod: <Error xmlns="http://schemas.microsoft.com/windowsazure"  
xmlns:i="http://www.w3.org/2001/XMLSchema-  
instance"><Code>ForbiddenError</Code><Message>The server failed to  
authenticate the request. Verify that the certificate is valid and is  
associated with this subscription.</Message></Error>
```

```
At line:1 char:1
```

```
+ Invoke-RestMethod $uri
```

```
+ ~~~~~
```

```
+ CategoryInfo          : InvalidOperation:  
(System.Net.HttpWebRequest:HttpWebRequest) [Invoke-RestMethod], WebException  
+ FullyQualifiedErrorId :  
WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeRestMethodCo  
mmand
```

Vidíte, že request nebyl správně ověřen oproti serveru. Musíme tedy přidat `CertificateThumbprint`. Ověřujeme certifikátem, který máme stažený v počítači (při prvotním nastavení PowerShell cmdletů).

```
PS C:\> Invoke-RestMethod -Uri $uri -CertificateThumbprint
C939114DA392E64947E4C22FB92FB92FB9
```

```
Invoke-RestMethod : <Error xmlns="http://schemas.microsoft.com/windowsazure"
xmlns:i="http://www.w3.org/2001/XMLSchema-
instance"><Code>MissingOrIncorrectVersionHeader</Code><Message>Request needs
to have a x-ms-version header.</Message></Error>
```

```
At line:1 char:1
```

```
+ Invoke-RestMethod -Uri $uri -CertificateThumbprint
C939114DA392E64947E4C22FB92FB92FB9
```

```
+
```

```
~~~~~
+ CategoryInfo          : InvalidOperation:
(System.Net.HttpWebRequest:HttpWebRequest) [Invoke-RestMethod], WebException
+ FullyQualifiedErrorId :
WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeRestMethodCo
mmand
```

Nyní vidíte, že chyba je odlišná. Jasně nám sděluje, že potřebujeme přidat onu verzi, kterou jsme viděli na jednom z předchozích obrázků. Přidáme tedy hash tabulku a uvedeme potřebné parametry. Hash tabulka by vás již neměla překvapit, používali jsme ji v několika minulých dílech.

```
PS C:\> Invoke-RestMethod -Uri $uri -CertificateThumbprint
C939114DA392E64947E4C22FB92FB92FB9 -Headers @{ 'x-ms-version'='2014-06-01' }
```

```
HostedServices
```

```
-----
```

```
HostedServices
```

Vidíme, že jsme opravdu dostali odpověď, která obsahuje nějaká (zatím neznámá) data. Vzhledem k tomu, že odpověď je standardní XML soubor, můžeme použít PowerShell syntaxi pro přístup k XML proměnné.

```
PS C:\> (Invoke-RestMethod -Uri $uri -CertificateThumbprint
C939114DA392E64947E4C22FB92FB92FB9 -Headers @{ 'x-ms-version'='2014-06-
01' }).HostedServices.HostedService.ServiceName
```

```
MakovecAZ083
```

```
MakovecAzATEST
```

```
mkutestvm1
```

```
PCfromtempl
```

Pokud vás zajímá, jaký tvar má odpověď, můžete se podívat na web, kde je vše podrobně popsáno. Malá část je zobrazena na následujícím obrázku.

The format of the response body is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<HostedServices xmlns="http://schemas.microsoft.com/windowsazure">
  <HostedService>
    <Url>address-of-cloud-service</Url>
    <ServiceName>name-of-cloud-service</ServiceName>
    <HostedServiceProperties>
      <Description>description-of-cloud-service</Description>
      <AffinityGroup>name-of-affinity-group</AffinityGroup>
      <Location>location-of-cloud-service</Location>
      <Label>label-of-cloud-service</Label>
      <Status>status-of-cloud-service</Status>
      <DateCreated>date-created</DateCreated>
      <DateLastModified>date-modified</DateLastModified>
      <ExtendedProperties>
        <ExtendedProperty>
          <Name>name-of-property-name</Name>
          <Value>value-of-property</Value>
        </ExtendedProperty>
      </ExtendedProperties>
      <ReverseDnsFqdn>reverse-dns-fqdn</ReverseDnsFqdn>
    </HostedServiceProperties>
    <DefaultWinRMCertificateThumbprint>certificate-thumbprint-for-winrm</DefaultWinRMCertificateThumbprint>
    <ComputeCapabilities>
      <VirtualMachineRoleSizes>
        <RoleSize>role-size-name</RoleSize>
      </VirtualMachineRoleSizes>
      <WebWorkerRoleSizes>
        <RoleSize>role-size-name</RoleSize>
      </WebWorkerRoleSizes>
    </ComputeCapabilities>
  </HostedService>
</HostedServices>
```

Element name	Description
Url	Specifies the request URI that is used to obtain information about the cloud service.
ServiceName	Specifies the name of the cloud service. This name is the DNS prefix name and can be used to access the service. For example, if the service name is MyService you could access the service by calling: http://MyService.cloudapp.net

Nyní máme seznam všech Services a můžeme se podívat na jednotlivé virtuální počítače, které jsou součástí těchto Services. Na MSDN opět najdeme formát pro dotaz:

<https://management.core.windows.net/<subscription-id>/services/hostedservices/<cloudservice-name>>

Pokud si přečtete informace na webu pozorně, což se při čtení na MSDN vždy hodí J, zjistíte, že URI může obsahovat parametr *embed-detail*, který (pokud je nastaven) vrátí více informací a Services. Pro náš dotaz jej tedy zapneme.

```
PS C:\> $uri= 'https://management.core.windows.net/<subscription-id>/services/hostedservices/mkutestvm1?embed-detail=true'
PS C:\temp> (Invoke-RestMethod -Uri $uri -CertificateThumbprint
C939114DA392E64947E4C22FB92FB92FB9 -Headers @{ 'x-ms-version'='2014-06-01' }).HostedService.Deployments.Deployment.Name
mkutestvm1
```

Vidíte, že výsledkem je jméno virtuálního počítače, který je v této Cloud Service. Pokud chceme replikovat funkcionalitu cmdletu **Get-AzureVM** můžeme si napsat jednoduchou funkci.

```
function Get-AzureVMfromREST
{
    $subscriptionID = '9a89542c-4657-807f-4fd0b2bd0b44'
    $certificateTP = ' C939114DA392E64947E4C22FB92FB92FB9 '
```

```

$requestHeader = @{ 'x-ms-version' = '2014-06-01' }

$uriHS =
"https://management.core.windows.net/$subscriptionID/services/hostedservices"

$uriVM =
https://management.core.windows.net/$subscriptionID/services/hostedservices/{0
}?embed-detail=true
(Invoke-RestMethod -Method Get -Uri $uriHS -CertificateThumbprint
$certificateTP -Headers $requestHeader).HostedServices.HostedService |
    ForEach {
        $hsName = $_.ServiceName
        Invoke-RestMethod -Method Get -Uri ($uriVM -f $hsName) -
CertificateThumbprint $certificateTP -Headers $requestHeader |
            ForEach {
                $prop = [ordered]@{
                    'ServiceName' = $_.HostedService.ServiceName
                    'Name' = $_.HostedService.Deployments.Deployment.Name
                    'Status' =
$_.HostedService.Deployments.Deployment.RoleInstanceList.RoleInstance.Instance
Status
                }
                New-Object -TypeName PSObject -Property $prop
            }
    }
}

```

A podíváme se na výsledek.

```
PS C:\> Get-AzureVMfromREST
```

ServiceName	Name	Status
-----	----	-----
MakovecAZ083	MakovecAZ083	StoppedDeallocated
MakovecAzATEST	MakovecAzATEST	StoppedDeallocated
mkutestvm1	mkutestvm1	StoppedDeallocated

Tip

Celý článek vznikl i proto, abych vám ukázal další pěkné použití proměnné `PSDefaultParameterValues` (také jsem o ní psal již dříve). Vzhledem k tomu, že momentálně nepřistupuji na jiné REST služby, než do své subskripce, může použít následující řádky pro nastavení mého prostředí.

```

# Azure REST API
$sID = '9a89542c-4657-807f-4fd0b2bd0b44'
$cTP = 'C939114DA392E64947E4C22FB92FB92FB9'
$HeaderRESTAzure = @{ 'x-ms-version' = '2014-06-01' }
$PSDefaultParameterValues.Add('Invoke-RestMethod:CertificateThumbprint', $cTP)
$PSDefaultParameterValues.Add('Invoke-RestMethod:Headers', $HeaderRESTAzure)
$PSDefaultParameterValues.Add('Invoke-RestMethod:Method', 'Get')

```

`PSDefaultParameterValues` definuje standardní hodnotu pro jednotlivé parametry definovaných cmdletů. Proto mohou předchozí příklad spustit nyní takto:

```

PS C:\> $uri =
https://management.core.windows.net/$sid/services/hostedservices
PS C:\> irm $uri

```

HostedServices

HostedServices

Parametry **Headers**, **CertificateThumbprint** a **Method** jsou automaticky doplněné.

Z dnešního článku byste si měli odnést poučení, které platí pro celý PowerShell. Pokud umíte pracovat v PowerShellu s jednou technologií a ovládáte jednotlivé části, můžete vaše znalosti přenést na úplně novou technologii (zdroj) bez větších problémů.

Přeji vám všem hezké skriptování v roce 2015.

HASH TABULKA

V PowerShellu se hash tabulka (nebudu zde používat české názvy hešovací tabulka, případně asociativní pole) používá relativně často. Existuje již od verze 1, ale časem se jednoduchost jejího použití vylepšovala. Vzhledem k tomu, že jsem v minulých dnech narazil nezávisle na její špatné použití, rozhodl jsem se, že vám dnes představím její použití.

Jak hash tabulka vypadá?

Základní použití je následující:

```
PS C:\> @{  
>> Jmeno = 'David'  
>> Prijmeni = 'Moravec'  
>> }  
>>
```

Name	Value
Prijmeni	Moravec
Jmeno	David

Celá struktura je uzavřena mezi znaky **@{ a }**. Uvnitř se používá dvojice klíče a hodnoty. K určitému klíči, zde např. **Jmeno** se přiřazuje hodnota, **David**. Vyhledávání se poté provádí na základě klíče.

Všimněte si jedné vlastnosti hash tabulky. V příkladu je vidět, že jsem zadal jako první klíč **Jmeno**, ale ve výsledku je pořadí klíčů prohozené. Hash tabulka nezajišťuje pořadí jednotlivých položek. Ve většině případů nám to nevádí, ale v určitých scénářích bychom zachované pořadí uvítali. Jak na to, si ukážeme dále.

Zkusme si hash tabulku přiřadit do proměnné.

```
PS C:\> $dm = @{  
>> Jmeno = 'David'  
>> Prijmeni = 'Moravec'  
>> Hobby = 'PowerShell'  
>> Zamestnani = 'Mainstream Technologies'  
>> }  
>>  
PS C:\> $dm
```

Name	Value
Jmeno	David
Hobby	PowerShell
Prijmeni	Moravec
Zamestnani	Mainstream Technologies

Vidíte, že použití je stejné jako u jiných datových typů. Hodnotu přiřadíme a dále použijeme. K jednotlivým položkám přistupujeme jako k vlastnostem objektu.

```
PS C:\> $dm.Jmeno
David
PS C:\> $dm.Zamestnani
Mainstream Technologies
```

Nebo pomocí „indexu“

```
PS C:\> $dm['Jmeno']
David
PS C:\> $dm['Zamestnani']
Mainstream Technologies
```

Do hash tabulky můžeme hodnoty přidávat:

```
PS C:\> $dm.Add('OblibenyNapoj','Kava')
PS C:\> $dm
```

Name	Value
-----	-----
Jmeno	David
Hobby	PowerShell
Prijmeni	Moravec
Zamestnani	Mainstream Technologies
OblibenyNapoj	Kava

Nebo je mazat:

```
PS C:\> $dm.Remove('Zamestnani')
PS C:\> $dm
```

Name	Value
-----	-----
Jmeno	David
Hobby	PowerShell
Prijmeni	Moravec
OblibenyNapoj	Kava

Použití

Existují zřejmě dva nejčastější způsoby použití. Při vytváření nových objektů a pro shromažďování informací. Při vytváření nových objektů se používá následujícím způsobem.

```
PS C:\> New-Object -TypeName PSObject -Property
@{Jmeno='david';Prijmeni='moravec'}
```

Prijmeni	Jmeno
-----	-----
moravec	david

Stejně bychom mohli použít již existující hash tabulku.

```
PS C:\> New-Object -TypeName PSObject -Property $dm
```

Jmeno	Hobby	Prijmeni	OblibenyNapoj
-----	-----	-----	-----
David	PowerShell	Moravec	Kava

Nyní si ukážeme, jak zajistit to, abychom vlastnosti měli v určeném pořadí. Jak jsem již říkal, v některých případech nám nejednoznačnost nevadí, ale zrovna v případě vytváření objektů se nám jasné pořadí vlastností hodí. Ve starších verzích PowerShellu jste museli použít následující trik:

```
PS C:\> New-Object -TypeName PSObject -Property $dm | Select-Object Jmeno,
Prijmeni, Hobby, OblibenyNapoj
```

Jmeno	Prijmeni	Hobby	OblibenyNapoj
-----	-----	-----	-----
David	Moravec	PowerShell	Kava

Po vytvoření objektu si zajistíme pořadí vlastností voláním cmdletu **Select-Object**. Od PowerShellu v3 můžeme použít tzv. seříděnou hash tabulku.

```
PS C:\> $dm2 = [ordered]@{
>> Jmeno = 'David'
>> Prijmeni = 'Moravec'
>> Hobby = 'PowerShell'
>> Zamestnani = 'Mainstream Technologies'
>> }
>>
PS C:\> $dm2
```

Name	Value
----	-----
Jmeno	David
Prijmeni	Moravec
Hobby	PowerShell
Zamestnani	Mainstream Technologies

Všimněte si rozdílu ve výstupu, vše je zapsáno tak, jak jsme chtěli. Nyní si můžeme vyrobit nový objekt.

```
PS C:\> New-Object -TypeName PSObject -Property $dm2
```

Jmeno	Prijmeni	Hobby	Zamestnani
-----	-----	-----	-----
David	Moravec	PowerShell	Mainstream Technologies

Nyní si ukážeme další použití hash tabulky. V určitých případech se nám hodí, že hash tabulka má pouze jediný klíč, ke kterému můžeme přiřazovat hodnoty. Nejčastějším použitím je vytváření různých statistik.

```
PS C:\> $h=@{}
PS C:\> cd Temp
PS C:\Temp> ls|%{$h[$_.Extension]=$h[$_.Extension]+1}
PS C:\Temp> $h
```

Name	Value
----	-----
.hta	1
.cmd	3
.PML	1
.png	15
.xsd	1
.html	4
.xml	12
.pssc	1
.clixml	1
.csv	12
.ps1	9

... zkráceno

Dokážete odhadnout, co jsme právě vyrobili? Nejprve jsme provedli výpis obsahu adresáře (poté, co jsme vytvořili prázdnou hash tabulku) a poté jsme do hash tabulky pro každý soubor uložili jeho příponu a ve výsledku spočítali celkový výskyt souborů určitého typu. Nyní můžeme výsledky dále zpracovat.

```
PS C:\Temp> $h | sort Value
```

Name	Value
.hta	1
.cmd	3
.PML	1
.png	15
.xsd	1
.html	4
.xml	12

...zkráceno

Vidíme, že standardním způsobem se nám třídění nepodařilo. Pro správné setřídění musíme použít metodu **GetEnumerator**, která nám zajistí přístup k jednotlivým položkám tak, jak očekává cmdlet **Sort-Object**.

```
PS C:\Temp> $h.GetEnumerator() | sort Value -Descending
```

Name	Value
	19
.png	15
.csv	12
.xml	12
.ps1	9
.txt	8
.exe	6
.html	4

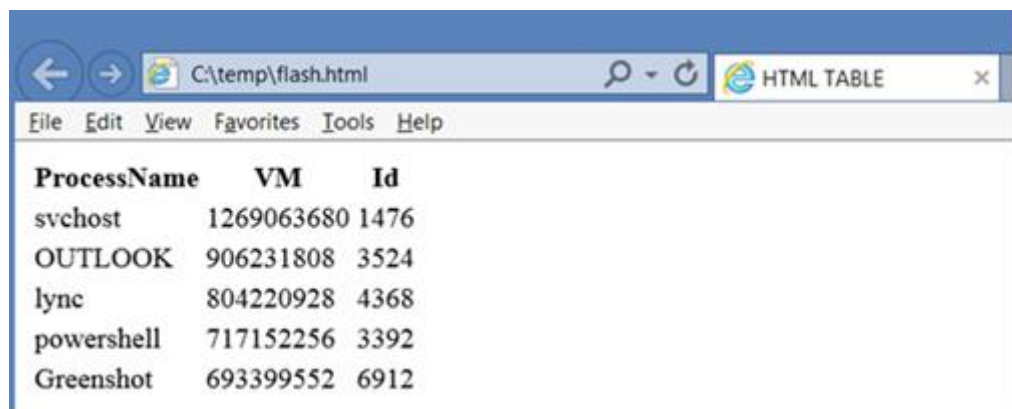
Nyní vidíme, že třídění proběhlo již podle předpokladů. Pokud by vás zajímalo více informací k hash tabulkám, můžete se podívat do velice pěkně zpracované nápovědy **about_Hash_Tables**.

EXPORT DO HTML

Poslední dobou pracuji více a více s reporty. Zákazníci evidentně chtějí reportovat svou práci a PowerShell je možností, jak si zkrátit čas potřebný pro tvorbu status mailů. Na školeních se mi stalo již několikrát, že následující (nebo podobný) one-liner způsobil u mých posluchačů údiv/úsmev/úžas/úlek]

```
PS C:\Temp> gps | Sort vm -desc | Select -First 5 -Property ProcessName, VM, Id | ConvertTo-Html | Out-File .\flash.html; ii .\flash.html
```

Výsledkem je otevření tohoto souboru:



Úmyslně jsem použil aliasy – zkušený čtenář toho Technet Flashe by měl být schopen většinu správných cmdletů identifikovat.

Dnes bych se chtěl věnovat cmdletu **ConvertTo-Html**. Tento cmdlet má několik zajímavých parametrů, které nám ulehčí tvorbu výsledného HTML kódu. Vše si ukážeme na následujícím krátkém kódu:

```
PS C:\temp> Get-Process -Name powershell | Select -Property ProcessName, Id, VM
```

```
ProcessName Id    VM
-----
powershell  7284 638828544
```

a výstup budeme konvertovat.

```
PS C:\temp> Get-Process -Name powershell | Select -Property ProcessName, Id, VM | ConvertTo-Html
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>HTML TABLE</title>
</head><body>
<table>
<colgroup><col/><col/><col/></colgroup>
<tr><th>ProcessName</th><th>Id</th><th>VM</th></tr>
<tr><td>powershell</td><td>7284</td><td>636764160</td></tr>
</table>
</body></html>
```

Pokud výstup uložíte do souboru, dostanete HTML soubor, který můžete dále sdílet. Pokud ovšem chcete výsledek cmdletu přidat jako součást většího reportu, hodí se následující parametr.

```
PS C:\temp> Get-Process -Name powershell | Select -Property ProcessName, Id, VM | ConvertTo-Html -Fragment
```

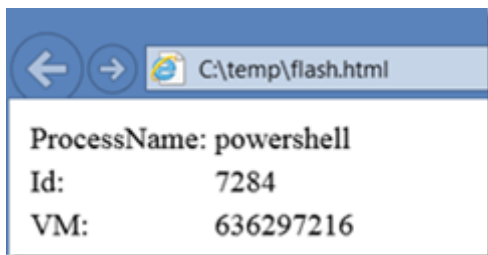
```
<table>
<colgroup><col/><col/><col/></colgroup>
<tr><th>ProcessName</th><th>Id</th><th>VM</th></tr>
<tr><td>powershell</td><td>7284</td><td>635437056</td></tr>
</table>
```

Vidíte, že z výsledného souboru se opravdu zapsal pouze fragment s výslednou tabulkou. Pokud bychom si tuto tabulku uložili do proměnné, můžeme je ve výsledku spojit dohromady a tím dosáhnout reportu s více tabulkami pod sebou.

Další možností je vložit výsledek jako list a nikoli jako tabulku.

```
PS C:\temp> Get-Process -Name powershell | Select -Property ProcessName, Id, VM | ConvertTo-Html -Fragment -As List
```

```
<table>
<tr><td>ProcessName:</td><td>powershell</td></tr>
<tr><td>Id:</td><td>7284</td></tr>
<tr><td>VM:</td><td>636502016</td></tr>
</table>
```

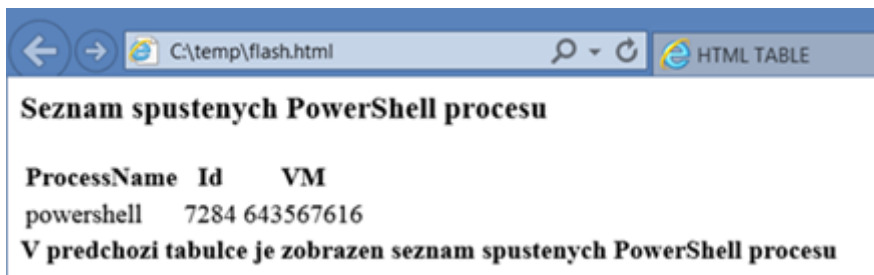


ProcessName:	powershell
Id:	7284
VM:	636297216

Jindy se vám bude hodit možnost přidat před (nebo za) výsledný report nějaký přidaný text. Toho lze dosáhnout pomocí parametrů **PreContent** a **PostContent**.

```
PS C:\temp> Get-Process -Name powershell | Select -Property ProcessName, Id, VM | ConvertTo-Html -Fragment
-PreContent '<h3>Seznam spustenych PowerShell procesu</h3>'
-PostContent '<b>V predchozi tabulce je zobrazen seznam spustenych PowerShell procesu</b>'
```

```
<h3>Seznam spustenych PowerShell procesu</h3>
<table>
<colgroup><col/><col/><col/></colgroup>
<tr><th>ProcessName</th><th>Id</th><th>VM</th></tr>
<tr><td>powershell</td><td>7284</td><td>643567616</td></tr>
</table>
<b>V predchozi tabulce je zobrazen seznam spustenych PowerShell procesu</b>
```



Seznam spustenych PowerShell procesu		
ProcessName	Id	VM
powershell	7284	643567616
V predchozi tabulce je zobrazen seznam spustenych PowerShell procesu		

V rámci těchto parametrů můžete použít HTML tagy a tím pádem ještě o trochu vylepšit grafickou reprezentaci zamýšleného reportu.

Pokud se podíváte do nápovědy pro cmdlet **ConvertTo-Html** jistě si všimnete i parametrů **Head**, **Title** a **Body**. Pomocí nich specifikujete částí HTML souboru.

Celá krása představeného přístupu vynikne ve spojení s cmdletem **Send-MailMessage**. Tento cmdlet má parametr, kterým říkáme, že tělo mailu je v HTML formátu. Proto můžeme předchozí report poslat mailem následující sadou příkazů. V proměnné **\$body** máme uložený výstup předchozího příkazu:

```
PS C:\temp> Send-MailMessage -Body $body -BodyAsHtml -To <mail> -SmtpServer
<server> -From <from> -Subject 'PowerShell procesy'
```

Pokud uvedeme správný parametr pro **SmtpServer**, bude mail poslán danému příjemci a v těle mailu bude uvedena tabulka z předchozího příkladu.

Doporučoval bych vám podívat se podrobněji na použití cmdletu **ConvertTo-Html**. Jeho využití je opravdu široké a sami nyní nevíte, kdy se vám může hodit.

Pokud by vás zajímalo, co se plánuje nového do PowerShellu v5, přijďte na konferenci [FRESH IT](#), kde se s vámi podělím o některé z plánovaných novinek. Ukážeme verzi z února tohoto roku, což je zatím

nejnovější release. Věřte, že je na co se těšit

POWERSHELL V5

Před pár dní jsem se zúčastnil konference FRESH IT a měl jsem možnost krátce prezentovat informace o novinkách v PowerShellu v5. Vzhledem k tomu, že téma se setkal s pochopením, rozhodl jsem se udělat v tomto článku krátký souhrn prezentovaných informací. Zároveň bych tyto informace rád v budoucnu rozšířil o podrobnější články tak, jak se bude blížit finální vydání nové verze.

PowerShell v5 lze nainstalovat jako součást balíčku nazvaného Windows Management Framework 5.0. Microsoft vydává update tohoto balíčku zhruba každé dva měsíce, zatím poslední verze je z února letošního roku. Můžete ji stáhnout na [tétu adrese](#). Aktuálně můžete tento update nainstalovat na Windows 8.1 a servery 2012 a 2012 R2. PowerShell 5.0 bude jako součást nainstalován na plánovaných Windows 10.

Pojďme si v krátkosti ukázat nejzajímavější novinky.

Transcript

Pokud používáte cmdlet Start-Transcript vězte, že od nové verze funguje i v ISE (případně jiných PowerShell prostředích). Přibyl nový parametr **IncludeInvocationHeader**, který do výsledného souboru přidává časovou značku spouštění jednotlivých příkazů.

```
*****
Command start time: 20150327173956
*****
PS C:\FRESH IT> get-date

27. března 2015 17:39:56
```

Symbolic links

Konečně jsme se dočkali cmdletů pro práci se symlinks. Vytvoření probíhá standardně pomocí cmdletu **New-Item**. Další z *-Item cmdletů obsahují parametry, které s linky pracují s typy **SymbolicLink**, **HardLink**, **Junction**. Položky můžete vytvářet, mazat, listovat.

Práce s archivy

PowerShell nově obsahuje cmdlety pro práci s archivy (ZIP). Dva cmdlety: **Compress-Archive** a **Expand-Archive** dokáží sbalit/rozbalit soubory tak, jak jste při práci s archivy zvyklí. Pro vytvoření archivu můžete pomocí parametru **CompressionLevel** určovat metodu a efektivitu algoritmu.

Zpracování nestrukturovaného textu

Již v prosincovém Flashi jsem psal o cmdletu ConvertFrom-String. Tento cmdlet umožňuje zpracovat složitěji formátovaný text tak, abyste z něj byli schopni získat objektovou reprezentaci. Podrobnější článek si můžete přečíst na [TechNet blogu](#).

OneGet a PowerShellGet

Tyto dva pojmy uslyšíte v budoucnosti velmi často. Dle mého názoru se jedná o asi nejzajímavější novinku v PowerShellu v5. Ve starších verzích to byl PS Remoting (v2), workflows (v3) a DSC (v4). Pomocí těchto dvou nástrojů budete schopni do PowerShellu instalovat SW (balíčky) a PowerShell moduly z dané repository (vlastní či externí). Představte si OneGet jako nástroj pro instalaci veřejně dostupných programů (např. 7-Zip nebo nástroje Sysinternals) a PowerShellGet jako nástroj pro instalaci modulů. Shrnutí na následujícím obrázku.

PowerShellGet cmdlety (module)

- Find, Install, Update, ...

OneGet cmdlety (package)

- Get, Find, Install, Uninstall, ...

Jádro OneGet

Jednotliví poskytovatelé balíčků

- PowerShellGet PSModule Provider
- NuGet Provider
- Chocolatey Provider

Obrázek čtete odspodu. Existují poskytovatelé balíčků. Tyto balíčky mohou být zpracovány pomocí nástroje OneGet. Nad OneGet sedí PowerShellGet, který funguje na stejném principu, ale pouze pro PowerShell moduly. Těmito nástroji můžete tedy po instalaci počítače nainstalovat potřebný software a nemusíte tuto instalaci řešit žádnými dalšími externími nástroji. Pokud zkusíte např.

```
Find-Package -Name zoomit | Install-Package -Verbose
```

Z určeného zdroje se najde balíček ZoomIt a jeho poslední verze se nainstaluje na váš počítač. Pokud znáte v Perlu ideu CPAN, tento princip bude nyní fungovat i pro PowerShell.

K těmto dvěma nástrojům se určitě dostanu v některém z dalších článků, protože si svou pozornost rozhodně zaslouží.

Mimochodem, dle posledních informací to vypadá, že nástroj OneGet bude brzy přejmenován na PackageManagement. Pokud by vás tedy toto téma zajímalo, zkuste vyhledávat obe termíny.

PSScriptAnalyzer

Toto je velmi zajímavý modul, který slouží ke statické kontrole vašeho kódu (skriptů a modulů). Microsoft připravil určitá pravidla a pomocí cmdletu **Invoke-ScriptAnalyzer** můžete oproti těmto pravidlům nechat zkontrolovat váš skript. Například jedna z pouček říká, že ve skriptu byste neměli používat aliasy, skripty budou lépe čitelné. Můžete tedy spustit pravidlo kontroly aliasů a na výstupu dostanete seznam, kde je vidět, kde jste se provinili proti tomuto pravidlu, např.

```
Invoke-ScriptAnalyzer -Path 'C:\FRESH IT\BlbySkript.ps1' -IncludeRule  
AvoidUsingCmdletAliases
```

Zobrazí seznam, kde vidíte své prohřešky a můžete je odstranit. Pravidel existuje několik a máte možnost si vytvářet i svá vlastní.

Script Debug

Z této oblasti vypíchnu asi nejzajímavější novinku – možnost provádět debug skriptu v jiné PowerShell session, než je aktuální. Celá „operace“ probíhá v několika krocích. Nejprve zjistíte ID procesu PowerShellu, kde chcete debug provést a připojíte se do něj.

```
Enter-PSHostProcess -Id $id
```

Zobrazíte si všechny Runspaces tohoto procesu.

```
Get-Runspace
```

Poté vyberete, kam se chcete připojit.

```
Debug-Runspace -Id 1
```

V tomto okamžiku jste připojeni do cizího procesu a jste schopni provést debug. Toto je rozhodně další z témat, které bych rád rozebral v samostatném článku.

Pro dnešek je to vše, ale věřte, že v PowerShellu v5 je opravdu několik skvělých novinek a je na co se těšit. Jak jsem již zmiňoval, k některým se vrátím později a s blížícími se Windows 10 bude těchto novinek publikováno stále více.

V [minulém díle](#) jsme se podívali na některé z novinek PowerShellu v5. Dnes povídání dokončíme a ukážeme si jednu novinku z oblasti Windows obecně.

PSScriptAnalyzer

Ano, pokud jste pozorní, pamatujete si, že o této části jsem již psal. Od doby napsání článku došlo ovšem ke změně. PSScriptAnalyzer byl přesunut jako samostatný projekt na GitHub (<https://github.com/PowerShell/PSScriptAnalyzer>). Zde si jej můžete stáhnout a nainportovat jako modul. Zároveň s tím bude v dalších verzích PowerShellu tento modul vyjmut jako standardní součást.

Desired State Configuration

V DSC došlo k asi největšímu počtu změn. K nejzajímavějším bych řadil především:

- Vytváření nových zdrojů (resources) pomocí tříd. Použití tříd (classes) v PowerShellu je také jednou z novinek v PowerShellu v5 a umožňuje více programátorsky laděným správcům využívat možnosti objektově orientovaného programování přímo v PowerShellu.
- Nyní můžete v ISE využít rozšířenou podporu pro práci s DSC (např. zvýrazňování syntaxe, doplňování, ...)
- Nové zdroje WaitForAll, WaitForSome, WaitForAny umožňují lépe sledovat závislosti mezi jednotlivými kroky. V následujícím obrázku vidíte např. v DSC jazyce ukázané použití WaitForAll při přidávání počítače do domény

```
configuration JoinDomain
{
    Import-DscResource -Module xComputerManagement

    WaitForAll DC
    {
        ResourceName      = '[xADDomain]NewDomain'
        NodeName           = 'MyDC'
        RetryIntervalSec   = 15
        RetryCount         = 30
    }

    xComputer JoinDomain
    {
        Name               = 'MyPC'
        DomainName         = 'Contoso.com'
        Credential         = (get-credential)
        DependsOn          = '[WaitForAll]DC'
    }
}
```

- **Get-DscConfigurationStatus** umožňuje zjišťovat aktuální stav dané konfigurace v jednotlivých nodech.
- **Compare-DscConfiguration** umí porovnat konfiguraci oproti jinému stavu, což přináší výhody při tvorbě a sledování DSC skriptů.

To byl výčet toho nejzajímavějšího z DSC. Opět jedno z témat pro budoucí články je podrobnější vhléd do DSC a využití všech novinek.

Cryptographic Message Syntax

CMS přináší možnost šifrování textu přímo v PowerShellu. Pokud máte certifikát typu “Document Encryption”, můžete v PowerShellu použít CMS cmdlety pro šifrování např. krátkého textu.

```
$protected = "Hello World" | Protect-CmsMessage -To "*me@somewhere.com*"
>> $protected

-----BEGIN CMS-----
MIIBqAYJKoZIhvcNAQcDoIIBmTCCAzuCAQAxggFQMIIIBTAIBADA0MCAxHjAcBgNVBAMMFw1ZWhv
bG1AbWljcm9zb2Z0LmNvbQIQYHsbcXnjIJCtH+OhGmc1DANBgkqhkiG9w0BAQcwAASCAQAnkFHM
proJnFy4geFGfyNmXH3yeoPvwEYzdnsoVqqDPA8D3wao77z70hJEXwz9GeFLnxD6djKV/tF4PxR
E27aduKSLbnxfpf/sepZ4fUkuGibnwWFrXGE3B1G26MCenHWjYQiqv+Nq32Gc97qEAERrhLv6S4R
G+2dJEnesW8A+z9QP0+DwYU5FzD0Td0ExrksWcKpLNR6j17Yaags3ltNVmbdEXekhi6Psf2MLMP
TS0791v2L0KeXFGuP0rdzPAwCkV0vNEqTEBeDnZGrjv/5766bM3GW34FXApod9u+VSFpBnqVOCBA
DVDraA6k+xwBt66cV840HLkh0kT02SIHMDwGCSqGSIB3DQEhATAdbglghkgBZQMEASoEEJbJaiRl
KMnBoD1dkb/FzSWAEBA8xkFwCu0e1ZtDj7nSjc=
-----END CMS-----
>> $protected | Unprotect-CmsMessage
Hello World
```

Nano Server

Před pár dny představil Microsoft nový operační systém, který je navržen pro běh v cloudu. Jmenuje se Nano Server a jedná se o minimalistickou (nano) verzi Windows Serveru. Pokud Server Core byl krokem k OS bez GUI, jde Nano Server ještě dál. Pokud se podíváme na velikost WMI souboru, uvidíme překvapivou velikost (malost?).

	Packages	5. 5. 2015 14:03	File folder	
	NanoServer.wim	25. 4. 2015 7:28	WIM File	142 065 KB
	ReadMe.txt	25. 4. 2015 7:28	Text Document	1 KB

Mělo by se jednat o OS, který bude mít rapidně menší množství updatů, rychlejší restarty a měl by být zaměřen převážně na dva scénáře: podpora cloudových aplikací a tvorba cloud infrastruktury. Nebudu zde ale zabíhat do marketingových čísel, informace si můžete najít sami. Co je zajímavé, je možnost administrace pomocí vzdáleného PowerShellu a WMI.

Nano Server můžete aktuálně stáhnout jako součást Windows 10 Server Technical Preview 2 (bohužel bylo potřeba opravdu stáhnout asi 4GB ISO, abych mohl získat 200MB) a dle návodu na stránkách Microsoftu si jej nainstalovat například na Hyper-V (viz aktuální strip Mistra Skriptika).

Po spuštění a připojení se na server, se můžete podívat na aktuální verzi PowerShellu.

```
[10.100.10.100]: PS C:\> $PSVersionTable
Name                           Value
----                           -
PSRemotingProtocolVersion      2.3
PSCompatibleVersions           {1.0, 2.0, 3.0,
4.0...}SerializationVersion    1.1.0.1
CLRVersion                     4.0.30319.34011
WSManStackVersion              3.0
PSVersion                      5.0.10074.0
BuildVersion                   10.0.10074.0
```

PowerShell je již ve verzi 5. Jak jsem již psal výše, administrace probíhá přes WMI (resp. CIM cmdlety), proto neexistuje například cmdlet `Get-Service`. Potřebné informace můžete ale samozřejmě získat CIM.

```
[10.100.10.100]: PS C:\> Get-CimInstance Win32_Service
```

ProcessId	Name	StartMode	State	Status	ExitCode
-----	----	-----	-----	-----	-----
1328	BFE	Auto	Running	OK	0
0	BITS	Manual	Stopped	OK	1077
0	ClusSvc	Disabled	Stopped	OK	1077
1204	CryptSvc	Auto	Running	OK	0
596	DcomLaunch	Auto	Running	OK	0
0	defragsvc	Manual	Stopped	OK	1077
596	DeviceInstall	Manual	Running	OK	0
1104	Dhcp	Auto	Running	OK	0
1452	DiagTrack	Auto	Running	OK	0
1204	Dnscache	Auto	Running	OK	0
1104	EventLog	Auto	Running	OK	0
0	IKEEXT	Manual	Stopped	OK	1077
0	KeyIso	Manual	Stopped	OK	1077
708	LanmanServer	Auto	Running	OK	0
1204	LanmanWorkstation	Auto	Running	OK	0
1104	lmhosts	Manual	Running	OK	0
1328	MpsSvc	Auto	Running	OK	0
0	Netlogon	Manual	Stopped	OK	1077
0	NetSetupSvc	Manual	Stopped	OK	0
1172	nsi	Auto	Running	OK	0
596	Power	Auto	Running	OK	0
708	ProfSvc	Auto	Running	OK	0
0	RemoteRegistry	Manual	Stopped	OK	1077
644	RpcEptMapper	Auto	Running	OK	0
644	RpcSs	Auto	Running	OK	0
0	sacsvr	Manual	Stopped	OK	1077
464	SamSs	Auto	Running	OK	0
708	Schedule	Auto	Running	OK	0
0	seclogon	Manual	Stopped	OK	1077
0	SmbWitness	Manual	Stopped	OK	1077

0	smphost	Manual	Stopped	OK	1077
0	swprv	Manual	Stopped	OK	1077
596	SystemEventsBroker	Auto	Running	OK	0
0	TieringEngineService	Manual	Stopped	OK	1077
0	TimeBroker	Manual	Stopped	OK	1077
0	TrustedInstaller	Manual	Stopped	OK	0
0	VaultSvc	Manual	Stopped	OK	1077
0	vmicguestinterface	Manual	Stopped	OK	1077
860	vmicheartbeat	Manual	Running	OK	0
884	vmicshutdown	Manual	Running	OK	0
1104	vmictimesync	Manual	Running	OK	0
0	vmicvmsession	Manual	Stopped	OK	1077
0	vmms	Auto	Stopped	OK	87
0	VSS	Manual	Stopped	OK	1077
1172	W32Time	Auto	Running	OK	0
0	WerSvc	Manual	Stopped	OK	0
1464	WinDefend	Auto	Running	OK	0
1172	WinHttpAutoProxySvc	Manual	Running	OK	0
708	winmgmt	Auto	Running	OK	0
1204	WinRM	Auto	Running	OK	0
0	wmiApSrv	Manual	Stopped	OK	1077
0	wuauserv	Manual	Stopped	OK	2147942450
0	wudfsvc	Manual	Stopped	OK	1077

To samé platí pro **Get-CimInstance Win32_Process**.

Pokud se podíváme na počet cmdletů, dostaneme číslo 184, což je o několik desítek cmdletů méně než v „normální“ verzi OS Windows. Seznam dostupných cmdletů naleznete v následující tabulce:

Add-Content	Get-ChildItem	Join-Path	Rename-ItemProperty
Add-History	Get-Command	Measure-Command	Resolve-Path
Add-Member	Get-Content	Measure-Object	Resume-Job
Clear-Content	Get-Credential	Move-Item	Select-Object
Clear-History	Get-Culture	Move-ItemProperty	Select-String
Clear-Item	Get-Date	New-Alias	Set-Alias
			Set-
Clear-ItemProperty	Get-Event	New-Event	AuthenticodeSignature
Clear-Variable	Get-EventSubscriber	New-Item	Set-Content
Compare-Object	Get-ExecutionPolicy	New-ItemProperty	Set-Date
Connect-PSSession	Get-FormatData	New-Module	Set-ExecutionPolicy
Connect-WSMan	Get-Help	New-ModuleManifest	Set-Item
ConvertFrom-Csv	Get-History	New-Object	Set-ItemProperty
ConvertFrom-			
StringData	Get-Host	New-PSDrive	Set-Location
Convert-Path	Get-Item	New-PSSession	Set-PSBreakpoint
		New-	
		PSSessionConfiguration	
ConvertTo-Csv	Get-ItemProperty	File	Set-PSDebug

Copy-Item	Get-ItemPropertyValue	New-PSSessionOption	Set-PSSessionConfiguration
Copy-ItemProperty	Get-Job	New-PSTransportOption	Set-StrictMode
Debug-Job	Get-Location	New-TimeSpan	Set-Variable
Debug-Runspace	Get-Member	New-Variable	Set-WSManInstance
Disable-PSBreakpoint	Get-Module	New-WSManInstance	Set-WSManQuickConfig
Disable-PSRemoting	Get-Process	New-WSManSessionOption	Sort-Object
Disable-PSSessionConfiguration	Get-PSBreakpoint	Out-Default	Split-Path
Disable-RunspaceDebug	Get-PSCallStack	Out-File	Start-Job
Disable-WSManCredSSP	Get-PSDrive	Out-Host	Start-Sleep
Disconnect-PSSession	Get-PSHostProcessInfo	Out-Null	Stop-Job
Disconnect-WSMan	Get-PSPProvider	Out-String	Suspend-Job
Enable-PSBreakpoint	Get-PSSession	Pop-Location	Tee-Object
Enable-PSRemoting	Get-PSSessionConfiguration	Push-Location	Test-ModuleManifest
Enable-PSSessionConfiguration	Get-Random	Read-Host	Test-Path
Enable-RunspaceDebug	Get-Runspace	Receive-Job	Test-PSSessionConfigurationFile
Enable-WSManCredSSP	Get-RunspaceDebug	Receive-PSSession	Test-WSMan
Enter-PSHostProcess	Get-UICulture	Register-ArgumentCompleter	Unblock-File
Enter-PSSession	Get-Unique	Register-EngineEvent	Unregister-Event
Exit-PSHostProcess	Get-Variable	Register-ObjectEvent	Unregister-PSSessionConfiguration
Exit-PSSession	Get-WSManCredSSP	Register-PSSessionConfiguration	Wait-Debugger
Export-Alias	Get-WSManInstance	Remove-Event	Wait-Event
Export-Csv	Group-Object	Remove-Item	Wait-Job
Export-FormatData	Import-Alias	Remove-ItemProperty	Wait-Process
Export-ModuleMember	Import-Csv	Remove-Job	Where-Object
ForEach-Object	Import-LocalizedData	Remove-Module	Write-Debug
Format-Custom	Import-Module	Remove-PSBreakpoint	Write-Error
Format-List	Invoke-Command	Remove-PSDrive	Write-Host
Format-Table	Invoke-Expression	Remove-PSSession	Write-Output
Format-Wide	Invoke-History	Remove-Variable	Write-Progress

Get-Alias	Invoke-Item	Remove-WSManInstance	Write-Verbose
Get-AuthenticodeSignature	Invoke-WSManAction	Rename-Item	Write-Warning

Nano Server je rozhodně jedno z témat, kterému se budeme někdy v budoucnu z pohledu PowerShellu věnovat. Jsem velmi zvědav na to, jak rychle se budou objevovat updaty a novinky pro tuto verzi OS.

WINDOWS 10 A POWERSHELL V5

V předchozích dílech jsme se podívali na novinky v PowerShellu v5. Jednalo se o zajímavosti napříč použitými operačními systémy. Dnes se podíváme na novinky, které souvisí přímo s novým operačním systémem – Windows 10.

Konzole

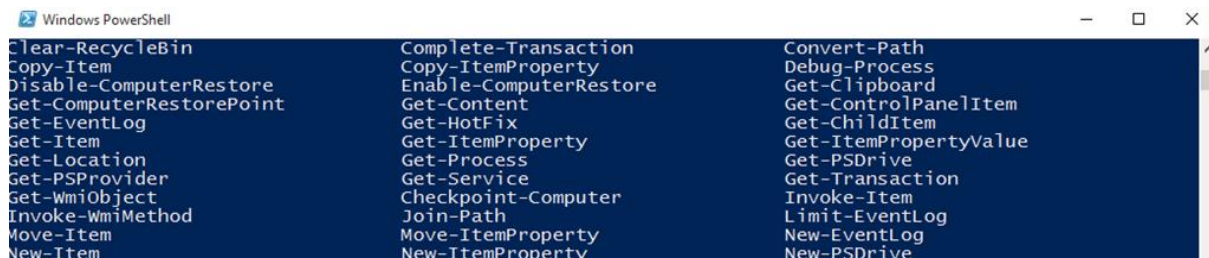
Již několik let se mezi uživateli PowerShellu skloňují žádosti o pořádný konzolový program. Ačkoli je PowerShell velice vyvinutým jazykem, konzole je stejné jako před několika lety.

Existují různé implementace – připomeňme si, že PowerShell host může být implementován do různých „konzolí – a o některých z nich jsme se bavili již dříve. Existuje například rozšíření **PsReadLine**, o které jsem také již psal. Všechny tyto kroky, ale stále nejsou ideálním stavem. Microsoft tuto „vadu“ odstanil právě ve Windows 10.

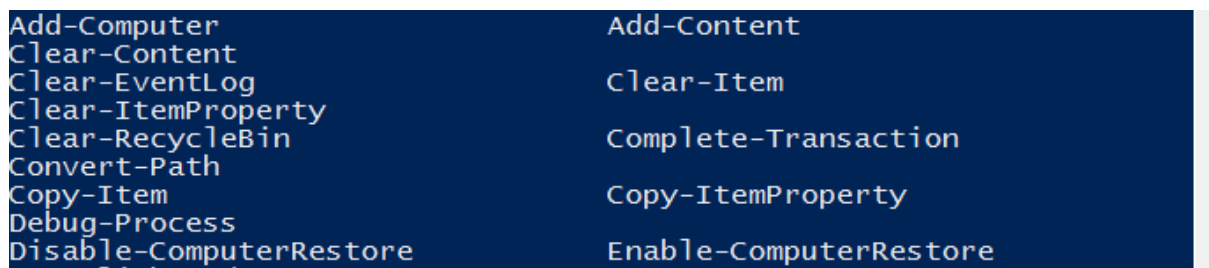
V následujících řádcích se podíváme na některé zajímavé novinky.

Zvětšení okna

V nových Windows již nemusíte měnit velikost okna pomocí menu **Vlastnosti**. Stačí myší zvětšit šířku a buffer je změněn automaticky. Stejně tak se překreslí obsah okna. Na následujícím obrázku je standardní velikost okna:



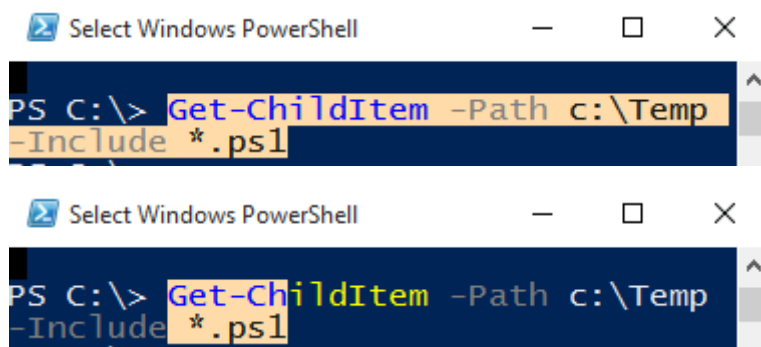
Pokud zmenším velikost okna myší, všimněte si, že se text automaticky přizpůsobuje nové velikosti okna:



Vybrání bloku textu

Velice častý požadavek na práci s textem v konzoli je vybrání určitého textu. V PowerShellu typicky napíšete příkaz, zjistíte, jestli funguje a pak si jej chcete překopírovat do schránky a použít v jiné

aplikaci. Standardní výběr textu prozatím probíhal v bloku, nikoli jako pokračování řádky. Porovnejte následující dva obrázky.



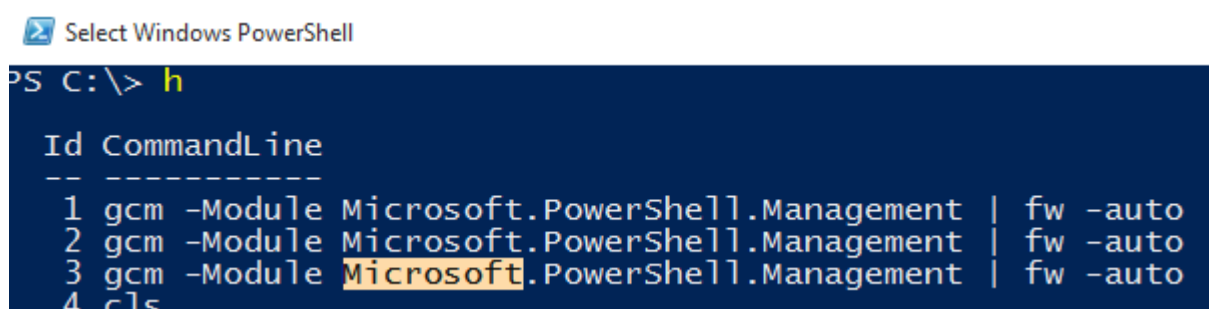
Na prvním vidíte nový způsob výběru, na druhém „starou verzi“. Pokud byste přesto chtěli vybrat blok textu, stačí při výběru stisknout klávesu **Alt**. Globálně pro konzoli lze tuto vlastnost vypnout ve vlastnostech okna:



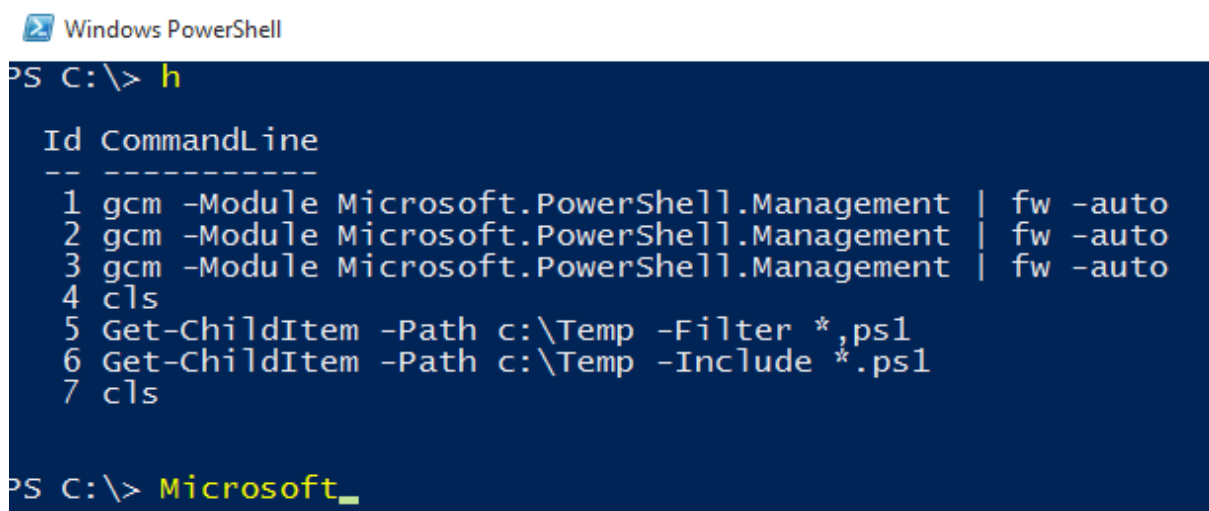
Výběr textu pomocí klávesnice

Práce s klávesovými zkratkami ve staré konzoli byla – mírně řečeno – tragická. Naštěstí lze v nové konzoli používat klávesové zkratky, na které jsme zvyklí z jiných aplikací.

Text můžete vybírat prostým použitím **SHIFT + šipky**, kdy například šipkou doprava byl označen text na dalším obrázku.

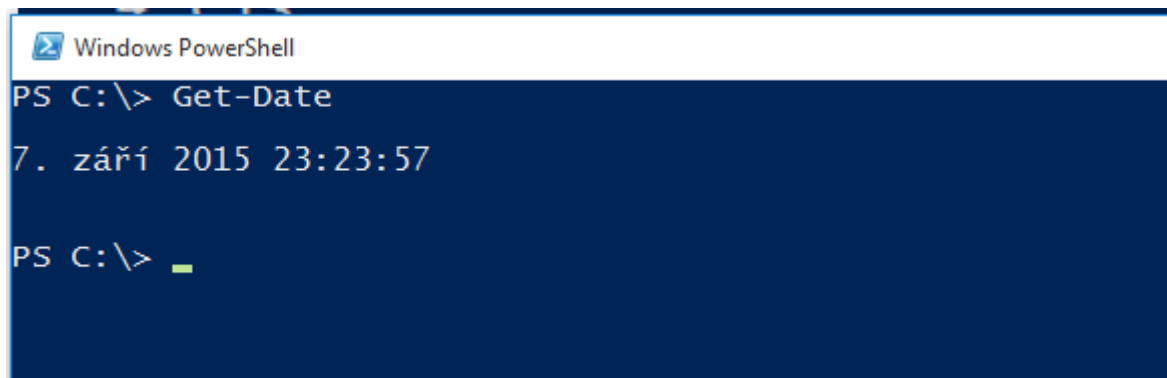


Pro editaci textu můžete použít zažité zkratky **Ctrl + C**, **Ctrl + V**, **Ctrl + X**. Na následujícím obrázku jsem pomocí Ctrl + C zkopíroval text označený na obrázku předchozím a poté jej pomocí Ctrl + V vložil na pozici kurzoru.



Další zajímavosti

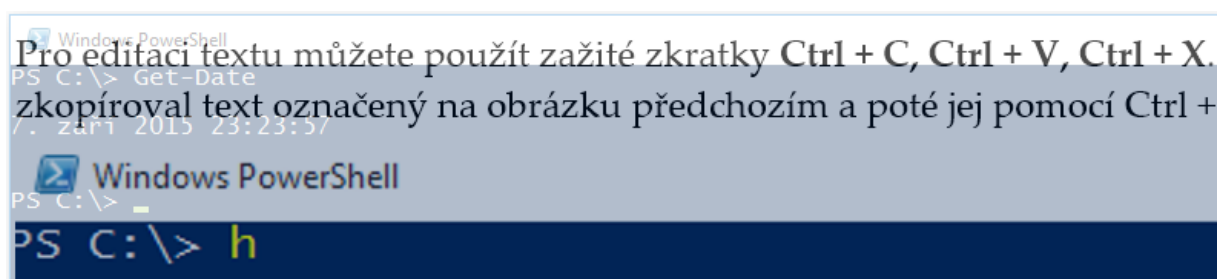
Jednou ze zajímavých možností je změna průhlednosti okna. Změna se může hodit, pokud potřebujete opsat nějakou část textu do konzole a nechce se vám přepínat mezi okny. Na následujících dvou oknech můžete porovnat nejvyšší a nejnižší povolenou průhlednost (minimální lze nastavit na 30%).



```
Windows PowerShell
PS C:\> Get-Date

7. září 2015 23:23:57

PS C:\> _
```



Pro editaci textu můžete použít zažité zkratky **Ctrl + C**, **Ctrl + V**, **Ctrl + X**. Zkopíroval text označený na obrázku předchozím a poté jej pomocí **Ctrl +**

```
Windows PowerShell
PS C:\> Get-Date

7. září 2015 23:23:57

Windows PowerShell
PS C:\> _

PS C:\> h
```

Pod konzolí mám otevřený Word s právě vznikajícím článkem. Každý si asi najde průhlednost, která mu vyhovuje. Experimentovat můžete pomocí kláves **Ctrl + Shift + Mínus (-)** a **Ctrl + Shift + Plus (+)**.

Ve spojení s použitým modulem PsReadLine je nová konzole velice dobrým pomocníkem při práci s PowerShellem. Kterou z novinek máte nejradši?

Vsuvka

Tento odstavec nesouvisí s výše zmíněným textem. Pouze bych rád opět zdůraznil nutnost porozumění helpu. Jeden z klientů mi poslal na kontrolu skript, který restartoval daný aplikační pool na IIS serveru. Skript našel kdesi na webu a jelikož si nebyl jistý, zda tento zhruba třicetřádkový kód dělá požadovanou akci, chtěl mít jistotu. Vstupem bylo jméno AppPoolu. Prolétl jsem skript, ale celé řešení mi přišlo trochu těžkopádné. Podíval jsem se do nápovědy k IIS a objevil následující cmdlet:

<https://technet.microsoft.com/en-us/library/ee790580.aspx>

Pokud pracujete s nějakou technologií – čtěte nápovědu jako první! Tot' pro dnešek vše. Přeji vám hodně zdaru při zkoumání novinek IT světa a PowerShellu zvlášť.

K dalšímu čtení doporučuji: www.logon.cz/tag/powershell/