Kapitola 3

Jazyk PowerShell

Vedle skutečnosti, že PowerShell je ovládacím příkazovým rozhraním čili shellem, se nám pod stejným jménem dostává do ruky především nový skriptovací jazyk jako takový. Nejde o žádnou „vylepšenou“ verzi jazyka jiného, jde o zcela nový jazyk, jehož kořeny jsou pevně zapuštěny v běhovém prostředí .NET Framework a jehož vlastnosti jsou určeny nesmírně moderním způsobem návrhu. Stačí jen málo přehánět, abychom řekli: vezměte to nejlepší z Perlu, Pythonu, jazyka C a Javy, zjednodušte to pro potřeby správců a zabudujte to do shellu. A právě to je nový jazyk PowerShell, jemuž se do hloubky budeme věnovat v této následující kapitole.

Konstrukce jazyka PowerShell

Aby mohl být jazyk příkazového rozhraní PowerShell používán jako opravdu plnohodnotné skriptovací prostředí, musí nezbytně nabízet základní konstrukce pro řízení běhu programu. A abychom my, administrátoři a tvůrci skriptů, mohli PowerShell účinně používat, potřebujeme se s těmito konstrukcemi seznámit, neboť bez nich budou naše skripty často neúčinné, nemotorné či případně vůbec nemožné.

V zásadě můžeme konstrukce pro řízení toku skriptu rozdělit na několik skupin, jimž se budeme věnovat v následujících podkapitolách. První dvě části řeší přímo otázku, kudy se bude skript dále ubírat a kolikrát se tak stane, další dvě části jsou pak věnovány členění skriptů do samostatných částí za účelem lepšího uspořádání. Ačkoliv možná mnohý čtenář prozatím usuzuje, že třeba problematika funkcí není až tak potřebná, pokusíme se zde ukázat, že právě zapojení síly funkce či jejího sourozence, jemuž tvůrci PowerShellu říkají filtr, je nesmírně prospěšným krokem vpřed při využití možností nového příkazového rozhraní.

Uzavíráme do bloků

V této úvodní části se pozastavíme u principu, který se může zdát stejně dobře naprosto samozřejmý (alespoň na první pohled) jako velmi těžko pochopitelný (při pohledech dalších, obzvláště při srážce s nějakým hodně záludným příkladem). Řekneme si něco blíže o tom, v jakých celcích vlastně PowerShell zpracovává příkazy, jež mu zadáváme, a jak lze tuto interpretaci ovlivnit.

PowerShell jako skriptovací prostředí pracuje v základním režimu, jejž bychom mohli nazvat také jako „zpracování funkčního řádku“. Výchozím předpokladem je, že pokud nepoužijeme speciální oddělovací či podobné konstrukce, je na jednom řádku jediný příkaz a vše ostatní jsou jeho parametry či přepínače. Následující příklad tento předpoklad podpoří:

```
76# Write-Host "Ted je:"
Ted je:
```

```
77# Get-Date
1. července 2007 0:02:16
78# Write-Host "Ted je:" Get-Date
Ted je: Get-Date
```

Ačkoliv oba příkazy pracují samostatně zcela bez problémů, jejich volné umístění do téhož řádku způsobí při interpretaci chování odlišné od toho, co bychom potřebovali. Prvním jednoduchým krokem je tedy oddělení příkazů: buďto můžeme respektovat pravidlo a funkčním řádku a příkazy umístit samostatně, nebo lze použít znak středníku a tím interpreteru naznačit, co je naším cílem:

```
81# Write-Host "Ted je:" ; Get-Date
Ted je:
1. července 2007 0:06:15
```

Pokud budeme naše skripty skládat z příkazů zapsaných na jednotlivých řádcích, narazíme na další přirozenou hranici, kterou může být celý zdrojový soubor – textový soubor s určitým počtem řádků, z nichž každý může obsahovat proveditelný příkaz s různými přepínači a parametry. Právě k možnostem volání skriptů v celých zdrojových souborech se ještě vrátíme později, nyní se však podíváme na další možnosti rozčlenění prováděcích řádků.

V jedné ze samostatných kapitol jsme si popsali geniální mechanismus roury a její vtělení do objektové podoby v PowerShellu. Roura je jeden ze způsobů, jak může být funkční řádek rozbit na části, jejichž provádění je odděleno do bloků s vlastními příkazy. Roura je oddělovač, na jehož obou koncích jsou pouze předávaná data – příkazy pracují na obou stranách roury samostatně a komunikují spolu právě zasíláním dat do roury. Proto je při interpretaci provádění příkazů jasně dáno:

```
87# Get-Process | ft wor* -AutoSize
WorkingSet WorkingSet64
---------- ------------
    139264       139264
    499712       499712
    552960       552960
(výpis pokračuje dle konfigurace systému)
```

PowerShell se řídí pořadím a nejdříve interpretuje příkaz na levé straně roury. Jeho výsledkem by měly být objektové proměnné, na něž roura čeká, a tato data jsou do ní zasílána. Příkaz na pravé straně od symbolu roury pak při interpretaci zpracuje vstupní data – nastupuje jako druhý v pořadí a k žádnému zmatení zde nedochází, ačkoliv pracujeme na jediném funkčním řádku. Příkazy jsou jasně odděleny, a navíc si v tomto případě díky synchronizaci způsobně předávají data.

Při práci s PowerShellem se však velmi často dostáváme do situací, kdy na rozčlenění jednotlivých prováděných akcí nestačí funkční řádky na jedné straně a hranice celého skriptu (ve formě zdrojového souboru) na straně druhé. Právě proto existuje způsob, jak vyčlenit určitou část skriptu a zajistit jí provádění v souvislém celku. Tato konstrukce je označována jako blok skriptu (scriptblok) a jde o nesmírně důležitý koncept, na nějž narazíme v řadě dalších konstrukcí i v této kapitole.

Pokud čtenář již prozkoumal řadu příkladů, na použití bloku skriptu jistě narazil. Rozpoznat jej můžeme podle oddělovače, kterým jsou složené závorky, a pokud PowerShell na takovouto konstrukci narazí, chová se k ní dle specifických pravidel. V první řadě je potře-

ba, aby celek byl uzavřen, neboť do té doby PowerShell nemůže nic interpretovat – zadáváme-li mu výslovně blok skriptu, musí jej vyhodnotit jako celek, o čemž se snadno přesvědčíme i v interaktivním režimu. Pokud totiž takovýto blok otevřeme, bude nás vyzývat k jeho uzavření:

```
91# {
>> Get-Process            [Stisk Enteru]
>>                        [Stisk Enteru]
>> }                      [Stisk Enteru]
>>                        [Stisk Enteru]
Get-Process
92#
```

Teprve po dokončení bloku přistoupil PowerShell k interpretaci, a zde možná nastalo malé překvapení. Přestože uvnitř je umístěn jinak normálně pracující příkaz, k jeho interpretaci nedošlo a namísto toho byl vypsán jako prostý řetězec. Pokud totiž vytvoříme blok skriptu, PowerShell očekává, že mu výslovně řekneme, co s ním má provést a jakou formu interpretace požadujeme. Možná to vypadá podivně, ale ve skutečnosti jde o zcela samozřejmou věc, jak uvidíme v mnoha příkladech typických situací v této knize. I když mírně předběhneme, pojďme si některé typické případy ukázat.

Jednou z možností, jak příkazový blok přimět k interpretaci, je výslovný operátor volání akce, zastupovaný znakem ampersandu. Následující příklad objasní rozdíl oproti výše uvedené situaci:

```
92# &{
>> Get-Date
>> }
>>
1. července 2007 0:42:40
```

Nyní je zřejmě na místě zcela logická otázka: proč bych předchozí operaci prováděl tak krkolomně? Odpověď je v zásadě prostá: blok skriptu se hodí všude, kde oddělení funkčních celků nelze jednoduše provést jinými způsoby, nebo kde potřebujeme provést určitou část kódu opakovaně, často s různými vstupujícími hodnotami. Velmi dobrou ukázkou je použití příkazu pro opakované zpracovávání všech objektů v rouře:

```
95# 1..10 | ForEach {$_ * 10}
10
20
30
40
50
60
70
80
90
100
```

V tomto případě je zde namísto znaku ampersand jako spouštěč cmdlet, jenž opakovaně říká PowerShellu, aby blok skriptu ve složených závorkách interpretoval (a navíc mu jako potravu předhazuje objekty z roury). Vezměme si jiný příklad z oblasti přímé práce s objekty:

```
Get-EventLog -LogName system | Where-Object -FilterScript {$_.entrytype -like "war*" }
```

Zde je blok skriptu dokonce parametrem pro přepínač samotného cmdletu – příkaz tedy řekne PowerShellu, že takto umístěný blok skriptu se má interpretovat jako výraz a jeho výsledek pak použit.

Na jiný dobrý příklad použití bloků skriptu narazíme velmi brzy při popisu podmíněných příkazů. V následující konstrukci se mohou vyskytovat dokonce bloky dva:

```
if ($user.psbase.InvokeGet("AccountDisabled") -eq $true)
        {
        write-host "Ucet je vypnut."
        }
else
        {
        write-host "Ucet je zapnut."
        }
```

Pokud rozumíme základům angličtiny, nemusíme ani znát onu konstrukci a přesto rozeznáme, že když bude cosi platit, bude použit jeden blok skriptu, a v opačném případě přijde na řadu blok druhý. Zde je tedy spouštěčem podmíněný příkaz, který říká, obsah kterého z bloků bude interpretován a kdy. Při řízení toku skriptů narazíme i na další konstrukce, jež v sobě oddělené bloky zahrnují jako přirozenou součást.

Abychom se v naší úvaze posunuli o krok dále, je třeba si uvědomit, že dosud byly bloky nepojmenované a jejich význam byl objasněn až souvislostmi, za kterých se „vyskytly“ – bez svých příkazů, jimž slouží, neměly smyslu. V PowerShellu (a zdaleka nejen v něm) však existují též bloky skriptu, kterým můžeme přiřazovat jména. Typickým příkladem je konstrukce, nazývaná funkce – na ni ještě narazíme a pro tentokrát si jen ukážeme, že jde vlastně také jen o blok, ovšem nějak pojmenovaný:

```
106# Function ctverec ($a) {
>> $a = $a * $a
>> $a
>> }
>>
107# ctverec 5
25
```

V tomto případě jsme učinili další krok k osvobození bloku skriptu – jeho volání již nezávisí na konkrétním umístění za tím či oním příkazem, je předepsán jako nezávislá činnost, která může být provedena kdykoliv voláním jména-identifikátoru. Umožňuje nám mnohem lépe navrhnout strukturu složitějších úloh a také zjednodušit řadu činností. Na tomto místě je důležité udělat opět další krok a uvědomit si, že jeden celý skript, uložený jako samostatný zdrojový soubor, je vlastně také jednou ucelenou funkcí. Způsob jeho ohraničení – zápis do odděleného zdrojového souboru – je jen jiným způsobem jak říci, kde samostatný blok začíná a kde končí.

Protože jsme si už ukázali různé situace, v nichž blok skriptu může hrát důležitou roli, zbývá prozradit poslední tajemství. Blok sám může být vnitřně ještě dále rozčleněn – to bychom asi očekávali a dovedeme si asi představit, že prostě dovnitř umístíme další složené závorky a oddělíme další kus kódu. V tomto případě však máme na mysli speciální, pojmenované části bloku skriptu, jež se posléze díky jasnému vymezení liší svým prováděním. Jinými slovy, tyto pojmenované části jsou PowerShellem interpretovány odlišně. Podívejme se na příklad, jenž celou věc demonstruje:


```
109# 10..15 | &{begin {"Start: $_"} process {$_ * 2} end {"Hotovo: $_"} }
Start:
20
22
24
26
28
30
Hotovo: 15
110#
```

Anglická pojmenování jsou opět poměrně výmluvná, nicméně popišme si chování bloku příkazů přesněji. Slova *begin, process* a *end* jsou klíčová, uvozují „své“ speciální bloky kódu a musejí být uvedena v dané podobě a pořadí. První a poslední část (*begin* a *end*) se vždy provádí jen jednou, kdežto část střední může být provedena jednou či opakovaně. Onen počet opakování je pak dán způsobem volání bloku – pokud blok skriptu zpracovává na vstupu objekty z roury, proběhne střední část právě tolikrát, aby všechny objekty z roury mohly být zpracovány. Pozor: toto chování nijak nenutí tvůrce skriptu, aby data z roury zpracoval, pouze to určuje počet opakování, jak ukazuje následující příklad:

```
110# 10..15 | &{begin {"Zaciname cist z roury: "} process {"Ted jsme zpracovali
objekt."} end {"Hotovo."} }
Zaciname cist z roury:
Ted jsme zpracovali objekt.
Ted jsme zpracovali objekt.
Ted jsme zpracovali objekt.
Ted jsme zpracovali objekt.
Ted jsme zpracovali objekt.
Ted jsme zpracovali objekt.
Hotovo.
```

V praxi je však samozřejmě mnohem častější situace, kdy objekty z roury opravdu zpracováváme, protože to má praktický užitek. Úplně na závěr uvedeme kompletní konstrukci bloku skriptu – v tuto chvíli bude možná příliš tvrdým oříškem, ale její použití bude ještě popsáno v dalších částech, takže následující příklad je možno chápat jako závdavek k dalšímu studiu. Vedle pojmenovaných částí lze totiž do nitra bloku jako celku ještě přesně vymezit parametry. To má smysl především u funkcí a celých skriptů v jednotlivých souborech, jak ukazuje následující příklad:

```
115# Function Block {
>> param ($vstup)
>> begin {"Zaciname:"}
>> process {Write-host $vstup}
>> end {"Koncime."}
>> }
>>
116# Block
Zaciname:
Koncime.
117# Block 25
Zaciname:
25
Koncime.
```

Na tomto místě popis bloků skriptu ukončíme, tedy alespoň dočasně: na jejich použití budeme dále narážet v řadě konstrukcí a samozřejmě také ve funkcích jako takových, na něž znovu narazíme na konci této části knihy.

Podmiňujeme

Konstrukce a příkazy pro větvení programů (skriptů) patří mezi nejsamozřejmější součásti programovacích jazyků i shellů a nemohou chybět ani v PowerShellu. Jak bývá i jinde dobrým zvykem, k dispozici jsou dvě koncepce větvení a tomu odpovídající dvě programové konstrukce, jež si ukážeme na následujících příkladech.

Podmínky If/ElseIf/Else

Tato podmíněná konstrukce je velmi tradičním a také důležitým prostředkem pro větvení skriptu či programu. Popsat ji můžeme následujícím předpisem:

```
If (<Rozhodujici_podminka>) {Prikazovy_blok_pri_platnosti} Else
{Prikazovy_blok_pri_neplatnosti}
```

Její způsob práce je pěkně vidět na následujícím příkladu:

```
85# $a = 100
86# $b = 50
87# if ($a -eq $b) {
>>   Write-Host "Hodnoty jsou si rovny"
>> } elseif ($a -lt $b) {
>>   Write-Host "Prvni je mensi nez druha"
>> } else {
>>   Write-Host "Prvni je vetsi nez druha"
>> }
>>

Prvni je vetsi nez druha
```

Za současného použití příslušných operátorů jsme roztřídili dvě hodnoty a odhalili jejich vztah. Podmínková konstrukce sestává za klíčovým slovem *If* z výrazu v obyčejných závorkách (ten rozhoduje o platnosti) a dále z programových bloků, jež čekají na provedení. Tuto část, věnovanou podmínečnému větvení skriptu, doplníme velmi hezkým příkladem, který opět (jako několik dalších úloh) pochází ze stránek Scriptcentra, kde byl uveden jako soutěžní úloha. Je výbornou ukázkou opakovaného nasazení konstrukce *If*, dále použití některých zajímavých operátorů a v neposlední řadě též chytrého využití objektového charakteru proměnných v PowerShellu. Konstrukce smyčky *For* je nám zatím neznáma, takže doporučujeme se k příkladu případně vrátit po nastudování následující kapitoly. I tak nám bude přinejmenším první část užitečná už nyní. Příklad vzápětí okomentujeme, takže na úvod pouze řekněme, že převádí římské číslice na arabské:

```
$a = Read-Host "Please enter a Roman numeral"
if ($a.contains("CM") -eq $True) {$intValue+= 900; $a = $a -replace("CM","")}
if ($a.contains("CD") -eq $True) {$intValue+= 400; $a = $a -replace("CD","")}
if ($a.contains("XC") -eq $True) {$intValue+= 90; $a = $a -replace("XC","")}
if ($a.contains("XL") -eq $True) {$intValue+= 40; $a = $a -replace("XL","")}
if ($a.contains("IX") -eq $True) {$intValue+= 9; $a = $a -replace("IX","")}
if ($a.contains("IV") -eq $True) {$intValue+= 4; $a = $a -replace("IV","")}
```

Takto umístěné podmínkové konstrukce pracují vlastně jako filtr. Příkazový blok za každou z podmínek pak zajišťuje práci počitadla: každý výskyt některé z porovnávaných hodnot znamená bod na její „konto“.

TIP Celou úlohu i s komentářem autorů naleznete na adrese: *http://www.microsoft.com/technet/scriptcenter/funzone/games/solutions07/apssol01.mspx*

Větvení pomocí Switch

Podmínečné větvení pomocí konstrukce s *If* má v řadě případů jednu zásadní nevýhodu, a to rozhodování typu ano/ne. Konstrukci lze samozřejmě opakovat a tím větvení dále rozvíjet, avšak v mnoha případech bychom ocenili přehlednější uspořádání a zápis. Právě pro tyto chvíle je v PowerShellu připraven příkaz podmíněného větvení s příznačným pojmenováním, pomocí nějž můžete váš skript přepnout na správnou „kolej“. Obecně vypadá předpis pro tuto konstrukci následovně:

```
$ridici_promenna
switch (<$ridici_promenna>)
        {
                Hodnota1 {<Blok_prikazu_1>}
                Hodnota2 {<Blok_prikazu_2>}
                Hodnota3 {<Blok_prikazu_3>}
                Hodnota4 {<Blok_prikazu_4>}
                Default  {<Blok_vychoziho_prikazu>}
        }
```

Ačkoliv první řádek do ní, striktně řečeno, ještě nepatří, zahrnuje konstrukce v podstatě vždy definici řídící proměnné, dle jejíhož obsahu se následovně bude rozhodovat. PowerShell poté testuje obsah řídící proměnné a v případě shody s vyjmenovanou hodnotou na tom kterém řádku provede související příkazový blok na témže řádku, kde shoda nastala. Velmi důležitá je poslední uvedená výchozí hodnota, která je zde jako pojistka pro případ, že předchozí testování selhala a nenastal ani jediný případ shody. Její přítomnost není nutná a často vyplývá z logiky úlohy, kterou řešíme.

Podívejme se však na konkrétní příklad, jenž spojí konstrukci *Switch* s několika dalšími prvky. Výpis skriptu dále podobně okomentujeme.


drives.ps1

```
$jednotky = [IO.DriveInfo]::GetDrives()
[int] $hdd = 0
[int] $externi = 0
[int] $cd = 0
[int] $sitove = 0
[int] $nezname = 0
foreach ($jednotka in $jednotky) {
$typ_jednotky = $jednotka.drivetype
if ($jednotka.isready) {
        switch ($typ_jednotky)
                {
                        "Fixed"  {$hdd+= 1}
                        "Removable"      {$externi+= 1}
                        "CDRom"  {$cd+= 1}
                        "Network"        {$sitove+= 1}
                        default          {$nezname+= 1}
        }
```

```
        }
}
Write-Host "Dostupne jednotky: `n HDD: ", $hdd, "`n Externi:", $externi, "`n CDROM:",
$cd, `
"`n Sitove:", $sitove, "`n Nezname:", $nezname
```

Úkolem skriptu je zjevně projít všechna připojená datová úložiště a provést jakousi inventarizaci. Používáme k tomu určité možnosti PowerShellu, jež budou popsány podrobněji v kapitole o .NET Frameworku (řádek 1), ovšem nás zajímá především konstrukce od klíčového slova *switch*. Řídící proměnná při běhu programu postupně vždy obsahuje typ připojené jednotky, jehož jména jsou rovněž dosti výmluvná, a na každém řádku naší větvící konstrukce je pak vložen příkazový blok přičítající body za každý jeden výskyt daného typu. Jinými slovy, po „prolustrování" všech jednotek máme stav počítadel v souladu s výskytem jednotlivých typů. Za povšimnutí určitě též stojí jediné podmíněné větvení *If*, jehož úkolem je vyloučit odpojené (nepřipravené) jednotky, jako je typicky nepřítomná disketa. Výsledek pak může vypadat třeba takto:

```
Dostupne jednotky:
 HDD:  2
 Externi: 1
 CDROM: 1
 Sitove: 2
 Nezname: 0
312#
```

Opakujeme

Bez příkazů opakování (cyklů) by byl PowerShell jen bezzubou, neživotaschopnou hříčkou a podobné mohou být i naše snahy o tvorbu pokročilejších konstrukcí, pokud se s cykly blíže neseznámíme. PowerShell opět nabízí několik typů konstrukcí podle toho, k čemu bude cyklus použit a jak budeme rozhodovat o jeho dalším průběhu.

TIP Některé typy cyklů jsou obzvláště vhodné pro práci s kolekcemi, tedy vlastně poli objektových proměnných. Právě využití cyklů při procházení a zpracování kolekcí je věnována jedna speciální kapitola této knihy, kde čtenář nalezne řadu dalších okomentovaných příkladů, jež prakticky naznačí využití konstrukcí cyklů.

POZNÁMKA V literatuře bývají jazykové konstrukce způsobující opakování též často označovány jako iterace.

Než si konstrukce opakování představíme podrobně jednu za druhou, nahlédneme do tabulky, která nám podává jejich přehled. Při jejím sestavování byla zohledněna tři kritéria: jedním z nich je umístění podmínky (před či za tělo cyklu) a s tím související minimální počet opakování (vůbec nebo vždy alespoň jednou), druhým způsob vyhodnocení podmínky samotné (můžeme na ni nahlížet pozitivně či negativně) a třetím pak znalost počtu opakování před spuštěním cyklu (smyčky vždy konečné a potenciálně nekonečné).

Konstrukce	Umístění podmínky	Platnost podmínky	Počet opakování
While	Začátek	Dokud platí	Nejistý
Do While	Konec	Dokud platí	Nejistý
Do Until	Konec	Dokud nenastane	Nejistý
ForEach		Dokud je prvek	Konečný
For	Začátek	Pevný počet	Konečný

Cykly příkazem While

Konstrukce cyklu uvozená klíčovým slovem While zde není zařazena na prvním místě náhodou. Jedná se o opakovací příkaz, jehož chování je poměrně intuitivní a konstrukce samotná není příliš složitá. Pokud bychom nahlédli do naší klasifikační tabulky, je tato konstrukce označena jako pracující s podmínkou na počátku, což naznačuje i obecný předpis:

```
while (<podmínka>){<příkazový blok>}
```

Z uvedené šablony je patrné, že po klíčovém slovu následuje podmiňující výraz a teprve po něm samotná prováděcí část – pokud nebude podmínka splněna, tělo zamýšleného cyklu se vůbec neprovede, což může být nanejvýš žádoucí. Podívejme se na první příklad, který je stejně „otřepaný“ jako výmluvný:

```
1# $pocitadlo = 1
2# while ($pocitadlo -le  10) { $pocitadlo; $pocitadlo++ }
1
2
3
4
5
6
7
8
9
10
3#
```

Povšimneme si především operátoru „menší nebo rovno“ (-le, less or equal), pomocí něhož je řídící proměnná konfrontována s hodnotou 10, a pak také způsobu, jakým je tato proměnná postupně zvyšována (inkrementována) v samotném těle cyklu.

POZNÁMKA Právě třeba možnost provádět takováto přičítání pomocí operátoru „++“ je významným přínosem jazyka PowerShellu oproti skriptování pomocí VBScriptu na platformě WSH. Tam takové konstrukce neexistují a docílit jich je možno třeba instalací podpory pro Perl, jenž vyniká nejen v těchto lahůdkách. PowerShell dozrál i v tomto směru.

Pojďme se přesvědčit, že podmínka na počátku cyklu opravdu funguje. Zkontrolujeme stav řídící proměnné a zkusíme cyklus „protočit“ znovu:

```
3# $pocitadlo
11
4# while ($pocitadlo -le  10) { $pocitadlo; $pocitadlo++ }
5#
```

Vše se zdá býti jasné: naše počítadlo je na stavu „11“ a pokus o další provádění cyklu je již v zárodku zlikvidován, neboť jsme neprošli přes úvodní podmínku. Nyní se podívejme na několik variací téhož příkladu, abychom se ujistili, že konstrukci správně rozumíme.

```
9# $pocitadlo = 0.1
10# while ($pocitadlo -le  5) { $pocitadlo; $pocitadlo++ }
0,1
1,1
2,1
3,1
4,1
11#
```

V předchozím případě je ukázáno, jakým způsobem probíhá přičítání řídící proměnné, následující ukázka zase naznačuje, že se jedná o cyklus, jehož počet opakování může být snadno (chybou tvůrce) nastaven na „nekonečno“:

```
21# $pocitadlo = 1
22# while ($pocitadlo -gt 0) { $pocitadlo; $pocitadlo++ }
1
2
3
(výpis stále pokračuje)
4350
4351
4352
(a stále pokračujeme)
```

Jinými slovy, jak říká naše tabulka: cyklus s podmínkou na počátku a dopředu neznámým počtem opakování.

Cykly příkazem Do While

Tento typ cyklu je vlastně pouze variací na předchozí téma, takže zachovává nejistotu v počtu opakování a přináší rozdíl v provedení vždy alespoň jedné akce před otestováním podmínky, jak naznačuje následující schéma:

```
do {<příkazový blok>} while (<podmínka>)
```

Z následujícího příkladu je patrné, že první „kolečko“, tedy přinejmenším jedno provedení těla cyklu proběhne za každou cenu, ačkoliv před prvním testováním je ve zjevném rozporu s následně testovanou podmínkou:

```
29# $pocitadlo = 1
30# do {$pocitadlo; $pocitadlo++ } while ($pocitadlo -lt 0)
1
31#
```

Cykly příkazem Do Until

Tento formát opakování je s předchozím velmi příbuzný a zásadním rozdílem je v této konstrukci pojetí podmínky a jejího vyhodnocení. Do pochopitelné řeči by se formulace podmínky dala přetransformovat zhruba v podobě rozkazu: prováděj tělo cyklu do té doby, než zjistím, že nastala testovaná situace (podmínka). Naznačme opět schéma:

```
do {<příkazový blok>} until (<podmínka>)
```

Na první pohled je patrné, že k testování dochází až na konci, takže tělo cyklu vždy proběhne alespoň jednou, ať je již kontrolní podmínka splněna nebo ne. Právě toto je patrné z následujícího příkladu:

```
42# $pocitadlo = 25
43# $pocitadlo
25
44# do { $pocitadlo; $pocitadlo++ } until ($pocitadlo -le 25)
```

Ačkoliv kontrolní proměnná – čítač – je zjevně pro podmínku vyhovující (již na počátku je roven hodnotě 25), tělo smyčky se provede přinejmenším poprvé, což ovšem způsobí osudovou chybu: počítadlo se okamžitě zvýší na hodnotu 26 a ani první, natož pak jakékoliv další testování už nemůže odvrátit nekonečný cyklus. Zkrátka, po prvním průběhu a prvním přičtení už řídící proměnná nikdy nenabude hodnoty menší či stejné jako 25.

Vyčíslování pomocí příkazu ForEach

Tato konstrukce využívající příkazu s velmi výmluvným názvem (asi bychom zkusili překlad „pro každou/každého/každé...“) je vůbec jedním z nejdůležitějších cyklů ve skriptovacích jazycích na platformě Windows. Patří mezi nejdůležitější i v prostředí WSH a objektově založený PowerShell její nezbytnost jenom podtrhuje. Slouží k postupnému procházení položek, jež jsou ukryty v datových strukturách typu pole či kolekce, a obzvláště onen druhý případ je velmi častý a žádoucí. Jak jinak, představíme si obecný předpis této smyčky:

```
foreach ($<položka> in $<kolekce>){<příkazový blok>}
```

Pokud bychom si probíhající akci měli popsat slovně, pak se děje asi toto: pro každou položku v kolekci, která je příkazu předložena, dojde k provedení stejného příkazového bloku, v jehož těle vystupuje aktuálně načtená položka z kolekce pod jménem první proměnné. Lépe patrné to bezesporu bude na konkrétním příkladu. Přestože řadu detailů v něm ještě neznáme, soustřeďme se na příkaz cyklu ForEach, o který nám momentálně jde:

sitovky_wmi.ps1

```
$strComputer = "."
$colItems = get-wmiobject -class "Win32_NetworkAdapterConfiguration" -namespace `
"root\CIMV2" -computername $strComputer -filter "IPEnabled = true"
foreach ($objItem in $colItems) {
      write-host "MACAddress: " $objItem.MACAddress
      write-host "IPAddress: " $objItem.IPAddress
      write-host
}
MACAddress:  00:53:45:00:00:00
IPAddress:  10.169.8.170
MACAddress:  00:50:56:C0:00:01
IPAddress:  192.168.199.1
MACAddress:  00:50:56:C0:00:08
IPAddress:  192.168.154.1
MACAddress:  00:03:C9:2D:BC:78
IPAddress:  0.0.0.0
```

První část skriptu přeskočme s tím, že „nějak“ načte vlastnosti síťových adaptérů na daném počítači a naplní příslušnou kolekci v proměnné $colItems. V cyklu pak celou kolekci položku po položce procházíme a pro každou jednotlivou položku ($colItem) vypíšeme její hardwarovou (MAC) adresu a síťovou adresu IP. Důležité je uvědomit si, jak je to s podmínkami a počtem opakování: podmínka jako taková by se dala formulovat výrazem

„dokud je co, tak to zpracovávej“ a počet opakování je předem jasně dán právě počtem položek v kolekci či poli. Ano, ačkoliv jej neznáme číslem, jistě víme, že je konečný a definitivní. Tedy, vlastně jej (skoro) známe a kdybychom chtěli (což si ukážeme vzápětí), můžeme se na něj zeptat. Tento příklad ukazuje, jak před spuštěním iterace můžeme zjistit, kolikrát vlastně proběhne:

```
48# $strComputer = "."
49# $colItems = get-wmiobject -class "Win32_NetworkAdapterConfiguration" ` -
namespace"root\CIMV2"
>> -computername $strComputer -filter "IPEnabled = true"
>>
50# $colitems.count
6
51#
```

Jak jsme již uvedli, patří tato konstrukce mezi vůbec nejpoužívanější.

Klasické cykly příkazem For

Jazyku PowerShell nechybí ani tradiční cyklus známý prakticky ze všech běžných programovacích jazyků. Vyznačuje se způsobem určení počtu opakování, neboť klasické pojetí rozhodovací podmínky, jež jsme viděli výše, je zde pozměněno na výslovné určení počtu opakování. Jinými slovy, příkaz For prostě tíhne k výslovnému nastavení počtu iterací. Zkusme naznačit nejdříve základní schéma:

```
For (<výchozí stav řícící proměnné> <podmínka> <změna řídící proměnné>) {<příkazový blok>}
```

Protože i v tomto případě je obecný předpis opravdu hodně schematický, porovnejme jej opět s příklady. A protože čísel už bylo až až, přidáme trošku písmen:

```
56# for ($i = 65; $i -le 75; $i++) { ([char] $i) }
A
B
C
D
E
F
```

Ano, vaše podezření je zcela správné, pohráli jsme si s číselnou reprezentací znaků v archaické, ale stále živé tabulce ASCII, v níž velké A má označení 65 a další písmena v abecedě následují. Z příkladu je dobře patrné, že od samého počátku je programová konstrukce v zajetí definitivního počtu opakování, jenž byl určen ihned v první chvíli. Za zmínku také stojí postup, pomocí nějž jsme převedli číslo na reprezentaci znakovou. Zopakujme příkazový blok ještě jednou:

```
{ ([char] $i) }
```

Tuto konstrukci nevidíme poprvé, a proto jen připomeňme, že se vlastně jedná o explicitní (jednoznačně určenou) datovou konverzi na proměnnou typu znak (pozor, neplést s proměnnou string, tedy delším řetězcem, do nějž se ukládají delší sekvence). Pokud je vstupní hodnota korektní (v našem případě celé číslo), konverze se podaří a vše funguje správně. Mimochodem, co se asi stalo v následujícím případě, že nám naše smyčka přestala fungovat?

```
100# for ($i = 120; $i -le 140; $i++) { ([char] $i) }
x
y
```

```
z
{
|
}
~

?
?
?
?
(výpis dále pokračuje poněkud monotónně)
```

V tomto případě již konverze selže, neboť jsme překročili „kapacitu“ proměnné typu *char*. Přesvědčit se o tom můžeme i jinak, když se pokusíme vnutit PowerShellu číslo s vysokou přesností pro účely námi požadované konverze:

```
63# for ($i = [double] 15.25; $i -le 25; $i++) { ([char] $i) }
Cannot convert value "15,25" to type "System.Char". Error: "String must be exactly one
character long."
At line:1 char:56
```

Co se přesně stalo? Desetinné číslo 15,25 je reprezentováno větším počtem bajtů a tyto nelze při automatické konverzi rozdělit tak, aby se z jednoho z nich stal znak tabulky ASCII. V příkladu výše to PowerShell udělal, jakmile přetekla kapacita, a způsobilo to samé znaky otazník „?“. Konverze „silou“ tedy proběhla, ale smyčka generovala nesmysly. V posledním příkladu již hrubá síla při konverzi nepomohla.

POZNÁMKA **Dvě poslední ukázky krásně demonstrují rozdíl mezi různými typy chyb v programech a skriptech. Druhá z nich je víceméně syntaktická – prohřešili jsme se proti pravidlům jazyka, operace nemohla proběhnout a interpreter nahlásil chybu, což je pro skriptaře záchrana a vysvobození. V příkladu předposledním chyba nenastala, PowerShell nic neřekl a provedl konverzi „silou“, takže následně vycházely nesmysly. Tyto chyby se označují jako logické a jsou-li o chloupek komplikovanější, dalo by se jim klidně říkat chyby pekelné, neb jejich odhalování představuje nesmírný intelektuální výkon.**

Na závěr kapitolky o cyklech si uveďme jeden delší příklad aplikace cyklu For, v němž bude naznačena jeho silná stránka: využití dopředu známého počtu opakování. Příklad byl převzat z webových stránek Scriptcentra (viz reference v dodatcích), kde se objevil jako řešení jedné ze soutěžních úloh. Ačkoliv je delší, není složitý a dále jej okomentujeme:

```
for ($a = 1; $a -le 4; $a++)
{
switch ($a)
    {
        1 {$operator1 = "+"}
        2 {$operator1 = "-"}
        3 {$operator1 = "*"}
        4 {$operator1 = "/"}
    }
    for ($b = 1; $b -le 4; $b++)
    {
    switch ($b)
        {
            1 {$operator2 = "+"}
            2 {$operator2 = "-"}
```

```
            3 {$operator2 = "*"}
            4 {$operator2 = "/"}
        }
        for ($c = 1; $c -le 4; $c++)
        {
        switch ($c)
            {
                1 {$operator3 = "+"}
                2 {$operator3 = "-"}
                3 {$operator3 = "*"}
                4 {$operator3 = "/"}
            }
            for ($d = 1; $d -le 4; $d++)
            {
            switch ($d)
                {
                    1 {$operator4 = "+"}
                    2 {$operator4 = "-"}
                    3 {$operator4 = "*"}
                    4 {$operator4 = "/"}
                }
            $strEquation = "12 " + $operator1 + " 8 " + $operator2 + " 4 " + $operator3
                            + " 2 " + ` $operator4 + " 9 "
         $answer = Invoke-expression $strEquation
            if ($answer -eq 23) {$strEquation}
            }
        }
    }
}
```

No a co tady úloha s několika cykly a krásnou ukázkou použití větvení pomocí přepínání vlastně dělá? Na jejím vstupu jsou známa čísla určitého matematického výrazu a také výpočetní výsledek a cílem je zjistit, jaké uspořádání operátorů takovýto výsledek dává. Čili hra „každý s každým“, což je lahůdka právě pro konstrukce větvení a opakování.

Vytváříme základní funkce

Ještě než čtenáře opustí poslední zbytky odhodlání zvládnout pokročilejší skriptovací techniky v PowerShellu, je třeba rychle říci: funkce jsou něčím, co je ve skutečnosti výrazně snazší a pochopitelnější, než jak se to jmenuje a na první pohled to vypadá! Překonáme-li první nástrahy sestavování delších skriptů se složitější logikou, stanou se totiž funkce něčím přirozeně potřebným. A protože opravdu nejsou ničím záhadným, ale naopak velmi užitečným konceptem, pojďme se podívat blíže, jak s nimi naložit.

Na úvod si řekněme, co to vlastně funkce je. Nejdříve stručně a výstižně: funkce je kus skriptu, který potřebujete zavolat ještě někdy příště, a proto je škoda jej zapomenout. Pokud budeme o něco exaktnější, pak je funkce pojmenovaný blok příkazů, jejž můžeme opakovaně spouštět právě oním jménem, jež jsme tomuto bloku přiřadili. No a na závěr definice dodejme, že hlavním vtipem funkce často je předání hodnot, jež budou „přechroupány“ právě pomocí příkazů v bloku uvnitř funkce. A nyní se pojďme podívat prakticky, co to vše vlastně znamená. První funkce bude opravdu jednoduchá:

```
9# function ted {get-date}
10# ted
16. března 2007 0:45:13
```

První řádek ukazuje hned několik důležitých vlastností. Funkce se definuje tím, že zapíšeme klíčové slovo *function*, za ně zařadíme námi zvolené jméno a posléze pomocí složených závorek vymezíme blok příkazů, jež budou funkci tvořit. Mezi těmito závorkami se pak může odehrávat velmi prostá operace, jako v tomto našem případě, kdy nevstupují do funkce žádné parametry. Na dalším řádku pak vidíme způsob volání funkce jejím jménem, následuje pak výstup činnosti funkce.

Přejděme k příkladu naprosto obligátnímu, totiž početnímu. Užitečnost v praxi je malá, ovšem pro názornost nám opět postačí:

```
7# function Vynasob {$args[0] * $args[1]}
8# vynasob 4 5
20
```

Úlohu jsme rozšířili o zpracování vstupních parametrů. Z příkladu je patrné, že při zavolání funkce jsou všechny vstupující hodnoty ukládány do speciální kolekce, pojmenované $args. V ní se budou všechny parametry nacházet a my je můžeme z této kolekce „vyzvednout" a použít, což provedeme poukázáním na daný prvek v kolekci pomocí jeho pořadového čísla (indexu). Jinými slovy, naše funkce postupně načetla první a druhou hodnotu a po vynásobení jsme ji obdrželi jako výstup z funkce.

Na pořadí vstupujících parametrů samozřejmě záležet může a nemusí, což dobře posoudíme na následujícím příkladu. Tentokrát bude naše funkce vstupující hodnoty spojovat a poukážeme zde také na další zajímavosti:

```
11# function SpojText {
>> $args[1] + $args[0] + $args[1]
>> }
>>
12# spojtext so te
tesote
13#
```

Začali jsme známým způsobem – klíčovým slovem a dále pojmenováním funkce a otevřením bloku příkazu. Z důvodu zpřehlednění jsme poté řádek ukončili klávesou Return (Enter), což PowerShell rozpoznal, a proto další řádek uvedl znaky šipek (>>) na znamení toho, že od nás očekává pokračování. Proto jsme dále funkci sestavovali a po ukončení řádku a další výzvě jsme celý blok uzavřeli ukončovací složenou závorkou. Přesně podle zadání pak bylo slovo poskládáno – druhý argument v pořadí stojí na prvním místě.

Funkce je možno vyrábět v interaktivním režimu, jako jsme to dosud prováděli my, nebo připravit předem a uložit v profilu uživatele – v tomto případě pak budou automaticky načteny při příštím spuštění prostředí. Pokud nás v kteroukoliv chvíli zajímá, jaké funkce jsou v PowerShellu definovány a jakými jmény je lze volat, můžeme se zeptat třeba tímto způsobem:

```
14# cd function:
Function:\
15# dir
CommandType     Name                                Definition
-----------     ----                                ----------
Function        prompt                              Write-Prompt ((Get-History
Function        TabExpansion                        param($line, $lastWord) Ge
Function        Clear-Host                          $spaceType = [System.Manag
Function        more                                param([string[]]$paths);
Function        help                                param([string]$Name,[strin
Function        man                                 param([string]$Name,[strin
```

```
Function        mkdir                          param([string[]]$paths); N
Function        md                             param([string[]]$paths); N
Function        A:                             Set-Location A:
Function        B:                             Set-Location B:
Function        C:                             Set-Location C:
Function        D:                             Set-Location D:
Function        E:                             Set-Location E:
Function        F:                             Set-Location F:
Function        G:                             Set-Location G:
Function        Vynasob                        $args[0] * $args[1]
Function        ted                            get-date
Function        Test-PscxPreference            param($name) if (Test-Path
Filter          Get-PropertyValue              param([string] $propertyNa
Filter          Remove-Accessors               process {…
```

Ve výpisu je dobře vidět, že vedle výchozích továrních nastavení (třeba funkce prompt) a dále funkcí zastupujících diskové jednotky pomocí tradiční písmenné konvence zde nalézáme i naše vlastí výtvory. Stejného výsledku se pak dobereme i pomocí jiných variant příkazů, jako je třeba tato:

```
16# Get-ChildItem function:*
```

Chcete-li se podívat nějaké funkci „na zoubek“, vyzkoušejte takovýto příkaz:

```
19# Get-ChildItem Function:SpojText | Format-List
Name        : SpojText
CommandType : Function
Definition  : $args[1] + $args[0] + $args[1]
```

Pokud chceme jakoukoliv existující funkci upravit, prostě ji znovu nadefinujeme (rozuměj přepíšeme) za použití jejího jména. Ukážeme si to na případu funkce *prompt*, jež ovlivňuje podobu výzvy, kterou nám PowerShell vrací na počátku příkazového řádku. Vytvořením různých variací je možno si výzvu upravit dle libosti:

```
28# function prompt {
>> "$(Get-Location)>`n`$ "
>> }
>>
Function:\>
$
```

A nebo třeba takto:

```
$ function prompt {
>>
>> Write-Prompt ((Get-History -Count 1).Id + 1)
>>     Update-HostTitle
>>
>>     return ' '
>> }
>>
30#
```

A pokud jste spíše minimalisté:

```
PS>function prompt {"=>"}
=>
```

Zajímavou a užitečnou konstrukcí v PowerShellu je speciální typ funkce, označovaný jako *Filter*. Jedná se vlastně o funkci, avšak vymezenou poněkud přesnějším způsobem: obsa-

huje pouze prováděcí blok příkazů, jenž slouží k transformaci objektů vystupujících z roury. Používá se pak třeba následujícím způsobem:

```
108# filter jmena_procesu {$_.processname}
109# Get-Process | jmena_procesu
AcroRd32
alg
ati2evxx
ati2evxx
atiptaxx
BTStackServer
BTTray
CooLSRV
```

Pokročilý vstup dat do funkcí

Protože již čtenář jistě ztratil zábrany a funkcí jako takových se nebojí, můžeme přistoupit k pokročilejším možnostem jejich využití. Při všech následujících ukázkách je vždy důležité si uvědomit, že se nijak nemění základní koncepce celé struktury: funkce je vždy kusem kódu skriptu, který je uzavřen v jeden celek pod určitým jménem pro snazší volání. A tomu uzavřenému, poměrně samostatně pracujícímu kusu skriptu lze předávat parametry ke zpracování, a to různě rafinovaně.

Řízení předávání parametrů

V předchozí části jsme viděli, že jednou z možností předání parametrů je výchozí kolekce *$args*. Mezi její výhody patří to, že prostě vznikne bez dalšího přičinění tvůrce skriptu vždy a pokud na vstupu funkce něco zadáme, dojde ke zpracování, pokud ovšem na prvky této kolekce ve skriptu nějak sáhneme. Na druhou stranu je tu nevýhoda v tom, že do takovéto kolekce může být vloženo cokoliv a při započetí zpracování funkce bychom měli provádět důslednou kontrolu, co že za prvky v této kolekci vlastně máme.

Funkce v PowerShellu jsou samozřejmě vybaveny prostředky i pro poněkud preciznější práci, což si právě ukážeme. Namísto výchozí kolekce lze využít výslovné pojmenování parametrů, jež do funkce vstoupí. Pak celá konstrukce může vypadat třeba takto:

```
20# function Jasny_vstup
>> {
>> param ($parametr01, $parametr02)
>> write-host $parametr01
>> write-host $parametr02
>> }
>>
21# Jasny_vstup 125 Ahoj
125
Ahoj
```

Výhodou takovéhoto zápisu je přinejmenším to, že při psaní pokročilejších konstrukcí uvnitř těla funkce budeme moci s jednotlivými parametry pracovat mnohem snáze, neb vhodně zvolená jména vstupujících parametrů jsou velmi dobrým vodítkem. Výše uvedená ukázka tedy naznačuje, že uvnitř těla funkce může být zahrnuta část *param*, jasně vyjmenovávající parametry tak, jak mají vstoupit dovnitř a být uloženy. Co se ale stalo s kolekcí *$args*? Podívejme se po ní v následující ukázce:

```
32# function Mizejici_params
>> {
>> param ($parametr01, $parametr02)
>> write-host "Pojmenovane parametry:"
>> write-host $parametr01
>> write-host $parametr02
>> write-host
>> write-host "Z kolekce $Args:"
>> foreach ($argument in $args)
>>    {
>>       write-host $argument
>>    }
>> }
>>
33# Mizejici_params 10 dvacet 30 40
Pojmenovane parametry:
10
dvacet
Z kolekce
30
40
34#
```

V naší funkci jsme tentokrát opět určili, že dva parametry na vstupu budou uloženy do pojmenovaných proměnných, a jak je vidět z následujícího výpisu, opravdu se tak stalo. Protože však na vstupu bylo zadáno parametrů více než očekávané dva, testujeme následně obsah výchozí kolekce *$args*, abychom viděli, jak se zachovala. Opět je z výpisu krásně vidět, že do této kolekce byly přiřazeny ty parametry, které přebývaly, tedy ony dva navíc – co nebylo uloženo do výslovně pojmenovaných proměnných, zpracoval PowerShell automaticky a uložil „pro strýčka příhodu".

Možnosti uložení vstupujících parametrů však tímto neskončily. Pokud už určujeme přímo jména proměnných, do kterých budou data na vstupu uložena, můžeme postoupit o krok dále a rovnou jim přiřadit datové typy. Opět se podívejme na ukázku:

fce_napevno.ps1

```
42# function Napevno
>> {
>> param ([int]$cele, [string]$retezec)
>> write-host $cele
>> write-host $retezec
>> }
>>
43# Napevno 10 20
10
20
```

V první řadě si povšimněme, že určení datového typu pro vstupující proměnné probíhá vlastně stejným způsobem, jako když deklarujeme běžné proměnné. Při prvním vyzkoušení se zdá být vše v pořádku a funkce pracuje, jak jsme zhruba očekávali. Podívejme se na jiné zadání:

```
44# Napevno 10 Dvacet 30
10
Dvacet
```

I tady se zdá být vše v pořádku – poslední hodnota není nijak zpracována, a proto ji ani nevidíme na výstupu, a první dvě hodnoty byly úspěšně uloženy a posléze funkcí vypsány do konzoly. Zkusme to jinak:

```
45# Napevno Deset Dvacet
Napevno : Cannot convert value "Deset" to type "System.Int32". Error: "Input string was
not in a correct format."
At line:1 char:8
+ Napevno  <<<< Deset Dvacet
```

Funkce nám konečně předvedla, v čem spočívá síla výslovného určení datového typu, a nepustila do svých „útrob" hodnotu, jež se ze vstupního parametru prostě nedokázala napasovat do určené kategorie. Tento mechanismus typové kontroly je samozřejmě užitečný úplně stejně, jako je tomu v jiných případech, kdy potřebujeme dbát na „typovou čistotu", abychom zabránili závažnějším chybám ve skriptech.

V předchozích příkladech jsme řešili situaci, kdy parametrů bylo oproti očekávání větší množství, avšak dosud jsme se nezastavili u situace zřejmě logičtější a častější: co funkce provede, když se jí parametrů nedostává? Nebudeme čtenáře tentokrát napínat a shrneme to: většinou se v lepším případě bude dít něco neočekávaného, v horším případě skončíme s chybou. Podívejme se na názorné příklady – v prvním případě budeme počítat, neboť naše funkce má vynásobit dvě vstupující hodnoty. Z ukázky je patrné, co se děje, když budou na vstupu chybět:

```
136# function Vynasob
>> {
>> param ([int]$cele, [int]$druhecele)
>> $vysledek = $cele * $druhecele
>>
>> write-host $vysledek
>> }
>>
137# Vynasob
0
138# Vynasob 10
0
139# Vynasob 10 20
200
```

PowerShell si prostě „nějak poradil" – v tomto případě se jednalo o dobře odhadnutelné chování, avšak u funkcí složitějších a skriptů poněkud propracovanějších na intuici rozhodně spoléhat nelze. Zkusme jiný příklad, k němuž využijeme funkci, popsanou zčásti i na jiném místě této knihy. Naším cílem bude zkontrolovat síťové adaptéry a najít nastavenou funkci přidělování konfigurace službou DHCP. Protože je to úloha častá, napsali jsme si funkci, a využijeme v ní schopnost WMI sahat na vzdálené počítače. Právě ono jméno počítače bude vstupním parametrem. Podívejme se, jak bude funkce pracovat:


check_
dhcp.ps1

```
140# function CheckDHCP {
>> param ($comp)
>> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=$true" -
ComputerName $comp
>> }
>>
143# CheckDHCP localhost
DHCPEnabled      : True
IPAddress        :
```

```
DefaultIPGateway :
DNSDomain        :
ServiceName      : NIC1394
Description      : 1394 Net Adapter
Index            : 2
(výpis pokračuje dle konfigurace stroje)
```

Naše jednoduchá funkce zjevně pracuje dle očekávání, pokud zadáme předpokládaný parametr. Zkusme na něj zapomenout:

```
144# CheckDHCP
Get-WmiObject : Cannot validate argument because it is null.
At line:3 char:97
+ Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=$true" -
ComputerName  <<<< $comp
145#
```

V tomto případě si již PowerShell nedokázal pomoci, protože bez určení cílového počítače naše konstrukce nemůže fungovat. Chybové hlášení je samozřejmě výmluvné, takže jistě nejpozději na druhý (až třetí) pokus funkci použijeme správně. Mohli bychom ji ale také vylepšit a zařídit, aby i v případě chybějícího vstupu proběhla nějaká výchozí akce. Proto naši funkci mírně upravíme:

check_ dhcp_ vylepsena. ps1

```
145# function CheckDHCP {
>> param ($comp=".")
>> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=$true" -
ComputerName $comp
>> }
>>
147# CheckDHCP
DHCPEnabled      : True
IPAddress        :
DefaultIPGateway :
DNSDomain        :
ServiceName      : NIC1394
Description      : 1394 Net Adapter
Index            : 2
```

V tomto případě jsou již problémy s chybou zažehnány a dosáhli jsme toho prostřednictvím výchozí inicializace vstupní proměnné. Jinými slovy řečeno, když na vstupu nic nebude (třebas i záměrně, pro zjednodušení), provede se výchozí přiřazení vstupní proměnné a funkce ji zpracuje, jakoby byla zadána přímo na vstupu. V našem posledním případě se tak do proměnné uloží znak tečky, jenž zastupuje lokální počítač, a pokud tedy nezadáme jméno jiného stroje v síti, obdržíme přehled nastavení lokálního operačního systému.

Co jsou filtry

V předchozích ukázkách jsme funkci v PowerShellu vždy chápali jako ucelený blok skriptu, který při zavolání obdrží parametry, provede své interní operace a vrátí výsledek. Je dobré si uvědomit, jak se PowerShell v takové chvíli chová – po zavolání funkce vše ostatní stojí, ona sama uvnitř zpracovává data, a teprve když je ukončena, data jsou zaslána zpět a vše dále pokračuje. Tento výchozí a jaksi předpokládaný způsob chování ovšem vždy není vyhovující, obzvláště proto, že PowerShell umí výborně pracovat s rourou. V čem je problém? Na výstupu z roury se objevují data opakovaně, tedy postupně jednotlivé prvky z větší kolekce. Co kdybychom si napsali pěknou funkci a chtěli ji zavolat na všechny

prvky takové kolekce na výstupu z roury? Vyzkoušejme to a použijme jednoduchou funkci, jež pro názornost bude pouze vypisovat vstupující hodnoty:

```
152# Function Block {
>> param ($vstup)
>> begin {"Zaciname:"}
>> process {Write-host $vstup}
>> end {"Koncime."}}
>>
153# Block 5
Zaciname:
5
Koncime.
```

TIP Pokud v tuto chvíli čtenář získal dojem, že jsme na něj připravili záplavu podivných novot, nechť nepropadá panice a pouze nahlédne zpět na začátek této kapitoly o řízení toku skriptu. Právě tam je obecná struktura bloku skriptu vysvětlena, včetně objasnění klíčových slov. Jejich význam bude navíc v dalších příkladech vzápětí patrný.

Naše nová funkce pracuje celkem dle očekávání – zpracuje vstupní hodnotu tak, že ji vrátí na výstup. Vyzkoušejme toto ještě jiným způsobem s tím, že se pokusíme předat větší množství hodnot. Pokusíme se zde k tomuto účelu využít prostředek nejpřirozenější, totiž rouru:

```
157# 1..5 | Block
Zaciname:
(volné řádky)
Koncime.
```

Nepodařilo se, PowerShell takto přímo parametry z roury do funkce nepředal. A pokud ano, pak o tom alespoň nevíme, protože se nám je nepodařilo správně uchopit. Zkusme jinou cestu – postupný výčet pomocí příslušného příkazu:

```
159# 1..5 | foreach {Block $_}
Zaciname:
1
Koncime.
Zaciname:
2
Koncime.
Zaciname:
3
Koncime.
Zaciname:
4
Koncime.
Zaciname:
5
Koncime.
160#
```

Ačkoliv jsme se dostali o krůček dále a akce probíhá téměř dle očekávání, je vidět, v čem tkví problém: funkce je volána opakovaně úplně celá, od počátku až do konce, včetně všech částí. Velkou nevýhodou je, že tak nemůžeme použít úvodní a závěrečné akce – pokud je ovšem z funkce odstraníme, vše může pracovat dle očekávání:

```
162# Function Block {
>> param($vstup)
>> process {Write-host $vstup}
>> }
>>
163# Block 5
5
164# 1..5 | foreach {Block $_}
1
2
3
4
5
```

Tímto „nuceným výsekem“ jsme si jistě pomohli, ovšem přišli jsme o dosti zásadní schopnost funkce provádět některé operace jen jednou. Představme si kupříkladu, že bychom chtěli, aby funkce nejen počítala s daty na výstupu z roury, ale ještě udržovala přehled o průběhu akce. Vytvoříme si tedy malé počítadlo a vzápětí bude jasné, kde je klíčová nevýhoda:

```
167# Function Block {
>> param($vstup)
>> begin {$pocitadlo = 0}
>> process {
>> Write-host $vstup
>> $pocitadlo = $pocitadlo + 1
>> }
>> end {write-host "Pocet opakovani: $pocitadlo"}
>> }
>>
168# Block 5
5
Pocet opakovani: 1
169# 1..5 | foreach {Block $_}
1
Pocet opakovani: 1
2
Pocet opakovani: 1
3
Pocet opakovani: 1
4
Pocet opakovani: 1
5
Pocet opakovani: 1
```

Je jasné, že takto počet prvků z roury prostě nesečteme – funkce při každém zavolání proměnné počítadla inicializuje a vypočítává znovu. My bychom však potřebovali něco jiného – funkci, která provede úvodní akci, pak jakoby nechá protéct data z roury a na závěr provede zakončovací akce, samozřejmě jen jednou. Jak na to?

V PowerShellu existují přinejmenším dvě cesty, jak problém uchopit. Zůstaneme-li u naší funkce, pak je potřeba změnit způsob, jak přijímá parametry. Pomůže nám v tom speciální proměnná PowerShellu, která automaticky předává celou kolekci prvků z roury do těla funkce a umožňuje její zpracování. Podívejme se na příklad a následně vše objasníme:

```
170# Function Block {
>> begin {$pocitadlo = 0}
```

```
>> process {
>> foreach ($hodnota in $input) {Write-host $hodnota}
>> $pocitadlo = $pocitadlo + 1
>> }
>> end {write-host "Pocet opakovani: $pocitadlo"}
>> }
>>
171# Block 5
Pocet opakovani: 1
172# 5 | Block
5
Pocet opakovani: 1
173# 1..5 | Block
1
2
3
4
5
Pocet opakovani: 5
```

Příklad naznačuje hned několik důležitých skutečností. V první řadě, našeho cíle je možno dosáhnout, funkce může odděleně zpracovat úvodní či závěrečnou část a pak jednotlivé prvky z roury. Dále platí, že tyto prvky jsou vždy uloženy ve výchozí proměnné (kolekci), jejíž jméno je *$input*, a jako kolekci ji tedy musíme procházet prvek za prvkem. Dále je vidět, že tato činnost pracuje jen tehdy, je-li vstup načten z roury. V neposlední řadě zdůrazněme, jak ke všemu došlo: nejdříve bylo vše vysypáno z roury do oné kolekce, pak byla funkce puštěna jen jednou a provedla potřebné akce.

Popsané řešení mám pomohlo, ale má zase jinou nevýhodu: než funkce dokončí svou práci, nikam dále se nedostaneme a s výsledky její práce nelze nic dělat. Což moc nepřeje skvělé koncepci zřetězení příkazů pomocí roury, jak naznačuje následující dvojice příkladů. První je zcela bez problémů, druhý je díky funkci nepoužitelný:

```
176# 1..5 | foreach {$_ * 2}
2
4
6
8
10
177# 1..5 | Block | foreach {$_ * 2}
1
2
3
4
5
Pocet opakovani: 5
```

Funkce je prostě jednorázová, nezpracovává výsledky průběžně a tyto nevystupují do roury. V PowerShellu se naštěstí nachází konstrukce, která dovoluje toto omezení překlenout. Jedná se o filtry a tyto jsou vlastně pouze speciálními případy funkcí – speciální jsou právě v tom, že zpracovávají rouru průběžně. Filtr nepracuje s žádnou kolekcí *$input*, jako tomu bylo u funkce, ale využívá běžnou proměnnou *$_* pro načítání objektu z roury. Poté provede akci a předá data dále k případnému následnému zpracování.

```
193# Filter Block {$_ * 2}
194# 1..5 | Block | foreach {$_ * 2}
```

```
4
8
12
16
20
```

Náš první filtr byl spíš názorný, než užitečný – zkrátka vstupní hodnotu zdvojnásobil. Poté byl výsledek zaslán znovu do roury, takže jsme mohli průběžně data odebírat a opět je násobit, tentokrát za použití jinak běžně známé konstrukce. Samozřejmě jsme mohli zvolit i toto řešení:

```
195# 1..5 | Block | Block
4
8
12
16
20
196# 1..5 | Block | Block | Block
8
16
24
32
40
```

Filtr tedy nijak nebrzdí proud a umožňuje další zpracování v rouře. Je to tedy jakýsi, pardon za ten výraz, „průtokový pozměňovač" – doufáme, že toto označení napomůže objasnit jeho práci.

POZNÁMKA Ačkoliv to nyní možná tak nevypadá, koncepce filtru je nesmírně významná a pro skriptování cenná. Jeho prostá schopnost provést stejnou operaci na každém prvku, který prostě načteme z roury, dovoluje provádět širokou škálu modifikací a dosahovat tak jinak velmi obtížně proveditelné akce. Jde o jeden z nejzásadnějších principů skriptování vůbec, a to nejen v PowerShellu – zde došlo k vylepšení díky objektové rouře.

Filtr může provádět i užitečné akce, jež přímo nemodifikují data z roury. Jedním z nejpěknějších případů je následující jednoduchý filtr, jejž autor převzal ze stránek tvůrců PowerShellu (a i oni se inspirovali na stránkách jiného „fanouška"). Jeho práce je patrná teprve ve chvíli, kdy jej opravdu spustíte, neboť z prostého výpisu výsledku nic nepoznáme:


filter_pomalu.ps1

```
199# filter pomalu ($tempo=100) { $_; Start-Sleep -milliseconds $tempo}
201# Get-Service | pomalu
Status   Name               DisplayName
------   ----               -----------
Stopped  Adobe LM Service   Adobe LM Service
Stopped  Alerter            Výstrahy
Running  ALG                Služba brány aplikačního rozhraní
Stopped  AppMgmt            Správa aplikací
(výpis dále pokračuje poměrně volným tempem)
```

Pozorný čtenář si možná říká, k čemu jsou nám filtry, když přeci máme konstrukci *ForEach*, která dokáže kolekce na výstupu z roury načítat a poté zpracovávat. Důvod je prostý – abychom nemuseli tuto konstrukci vytvářet na mnoha místech znovu, prostě si vyrobíme její pojmenovanou variantu. A to je vlastně celá podstata filtru – principem jednoduché, ale možnostmi využití nesmírně cenné konstrukce.