UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

# Analyzátor konfigurace počítače s využitím PowerShell

Jiří Kratochvíl

Bakalářská práce
2013

**Univerzita Pardubice**
**Fakulta elektrotechniky a informatiky**
Akademický rok: **2012/2013**

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Jiří Kratochvíl** |
| Osobní číslo: | **I10099** |
| Studijní program: | **B2646 Informační technologie** |
| Studijní obor: | **Informační technologie** |
| Název tématu: | **Analyzátor konfigurace počítače s využitím PowerShell** |
| Zadávající katedra: | **Katedra informačních technologií** |

Zásady pro vypracování:

Autor práce představí principy a možnosti využití scriptovacího jazyka PowerShell a porovná jej s prostředím Bash vybrané linuxové distribuce. V praktické části autor naprogramuje analyzátor konfigurace počítače s operačním systémem Windows 7 s využitím předdefinovaných a vlastních skriptů vytvořených v jazyce PowerShell. Pomocí PowerShellu představí nové možnosti správy a konfigurace systémů Windows s využitím všech nástrojů příkazového řádku, konzistentní syntaxí a mnoha doplňky. Výstup bude zobrazován v grafické aplikaci vytvořené v jazyce Java.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

**LEE, Thomas L. Windows powershell 2.0 bible. 1st ed. Indianapolis, IN: Wiley Pubishing, Inc., 2011, p. cm. ISBN 11-180-2198-3**
**MALINA, Patrik. Jak vyzrát na Windows PowerShell 2.0. Vyd. 1. Brno: Computer Press, 2010, 464 s. ISBN 978-80-251-2732-2**
**Windows Powershell 3.0 Step by Step. Microsoft Pr. ISBN 978-073-5663-398**

Vedoucí bakalářské práce: **Ing. Soňa Neradová**
Katedra softwarových technologií

Datum zadání bakalářské práce: **21. prosince 2012**
Termín odevzdání bakalářské práce: **10. května 2013**


L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

**Prohlášení autora**

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 3. 5. 2013 Jiří Kratochvíl

**Poděkování**

Chtěl bych poděkovat všem, kteří mě podporovali během mého studia i psaní bakalářské práce, zejména mé rodině, která mi vždy byla velkou oporou jak při studiu, tak v mém životě, a přítelkyni za její podporu, trpělivost a nesmírnou laskavost. Také bych chtěl poděkovat vedoucí mé práce, Ing. Soně Neradové, za její ochotu a cenné připomínky.

**Anotace**

Bakalářská práce se zabývá jazykem a programem PowerShell, zejména seznámením čtenáře s jeho použitím, popisem syntaxe a srovnáním s jazykem a programem Bash. Také představuje program, který je napsaný v jazyce Java a používá PowerShell pro konfiguraci počítače.

**Klíčová slova**

PowerShell, skript, konfigurace, syntaxe, Bash, jazyk, shell

**Title**

Computer configuration analyzer using PowerShell.

**Annotation**

Bachelor thesis deals with PowerShell language and shell, especially acquaints reader with its usage, description of syntax and comparison with language and program Bash. Also presents program, which is written in Java language and uses PowerShell for configuration of computer.

**Keywords**

PowerShell, script, configuration, syntax, Bash, language, shell

# Obsah

## Terminologie

Shell – Program s textovým rozhraním pro práci s operačním systémem. Často obsahuje i skriptovací jazyk pro usnadnění práce. [1]

Bash – „Shell a příkazový interpret pro operační systémy GNU.“ [2]

Interpret – „Program, který provádí instrukce vyššího programovacího jazyka.“ Tyto instrukce nejsou předem nijak kompilovány. [3]

CLI – Zkratka pro command line interface neboli rozhraní příkazové řádky. Jedná se o textové negrafické rozhraní. [4]

Cmdlet – Příkaz, který pracuje s objekty v PowerShellu. Vyslovuje se jako commandlet. [5]

.NET Framework – Framework pro tvorbu aplikací na systémy MS Windows. Obsahuje také prostředí .NET, které zprostředkovává běh aplikací, jejich správu paměti, vláken atd. Jedná se vlastně o virtuální stroj. [6]

Web-Based Enterprise Management – Standard pro správu operačních systémů. [7]

WMI – Zkratka pro Windows Management Instrumentation. Jedná se o implementaci Web-Based Enterprise Management, kterou vytvořila firma Microsoft. [8]

## Seznam obrázků



## Seznam tabulek

# Úvod

Správa počítače není triviální úloha. Často vyžaduje dobré znalosti výpočetní techniky a bývá problémem i pro zkušené uživatele. Obzvlášť velký problém je mít přehled o celém počítači na jednom místě. Zkušený správce systému může mít připravené skripty, které mu umožní získat během chvilky přehled o hardwaru i softwaru. Také pro něj není problém orientovat se v jednotlivých nastaveních systému. Problém však nastává, když chce mít přehled o počítači a zároveň měnit jeho různá nastavení. V systému Windows je zpravidla potřeba „proklikat se“ mnoha různými okny, než se člověk dostane k potřebnému nastavení počítače, nemluvě o uživatelích, kteří nemají hlubší znalost počítače.

Dlouhou dobu také nebyl na systémech Windows žádný opravdu užitečný prostředek pro správu systému administrátorem. Existoval sice příkazový řádek cmd.exe, ale svými schopnostmi v žádném případě nemohl konkurovat ani mnohem starším shellům ještě ze systémů Unix, potažmo ze systémů GNU/Linux. Ani Batch File Language, jazyk sloužící k psaní skriptů v systémech společnosti Microsoft, nemá takové možnosti jako například Bourne Shell. Navíc Batch File Language má velmi specifickou syntaxi nepříliš podobnou jiným skriptovacím jazykům. Naštěstí však firma Microsoft vytvořila nový shell a skriptovací jazyk, zvaný PowerShell. Ten má mnohem modernější syntaxi a poskytuje užitečné prostředky pro správu počítače. Právě tomuto jazyku se bakalářská práce věnuje.

První kapitola tohoto dokumentu se věnuje jazyku PowerShell, představuje ho z hlediska syntaxe a vysvětluje základy jeho použití. Ve všech těchto ohledech je také srovnáván se shellem a skriptovacím jazykem Bash.

Druhá kapitola pojednává o Windows Management Instrumentation, vysvětluje, co tento pojem znamená a ukazuje, k čemu je v systému použitý.

Třetí a poslední kapitola popisuje aplikaci PSConfig, která má za úkol usnadnit uživateli práci s počítačem z hlediska zjišťování informací o počítači a jeho konfigurace.

## Použité konvence

Většina oddílů první kapitoly, pojednávající o jazyku PowerShell, je rozdělena na dvě části. První obsahuje informace o PowerShellu a druhá o Bashi. V textu je také obsaženo mnoho ukázek použití těchto jazyků. K jejich odlišení jsou použity následující konvence.

**Tabulka 1 – Použité konvence**

| | |
|---|---|
| **$** | Prompt Bash shellu |
| **PS>** | Prompt PowerShellu |
| .ps1 | Koncovka souboru skriptu PowerShellu |
| .sh | Koncovka souboru skriptu Bashe |

Prompt je krátká zpráva, která oznamuje, že je požadován vstup uživatele. Označuje se tak řádek interaktivního shellu. Všechny příklady kódu v této práci představují buďto uživatelský vstup do interaktivní konzole shellu (tyto řádky začínají příslušným promptem) nebo skript (na začátku každého skriptu je v komentáři označeno, že se jedná o skript a je uveden název skriptu).

Zdrojový kód je graficky upraven tímto stylem:

```
Toto je blok zdrojového kódu {
…
}
```

# 1 Popis PowerShellu a porovnání s Bashem

Tato kapitola se věnuje představení programu PowerShell z hlediska základního použití a syntaxe jeho skriptovacího jazyka a porovnání s Bash shellem v Linuxu. Čerpá především ze zdrojů [9] [10] [11] [12] [13].

## 1.1 Představení PowerShellu

Windows PowerShell je shell s textovou příkazovou řádkou a zároveň skriptovací jazyk od firmy Microsoft. Jeho první verze byla vydána v roce 2006. Slouží ke správě systémů Windows od verze Windows XP, Windows Server od verze Windows Server 2003 a aplikací, které se v těchto systémech vyskytují. Správcům systému poskytuje nástroje pro manipulaci se systémem srovnatelné s utilitami operačních systémů GNU/Linux. PowerShell je postaven na frameworku Microsoft .NET a také plně podporuje Windows Management Instrumentation (WMI) třídy, které budou zmíněny dále v samostatné kapitole.

Shell PowerShellu je vlastně interpretem skriptovacího jazyka PowerShell a dal by se považovat za jistou kombinaci Bashe a interpretů moderních skriptovacích jazyků. Tím je myšleno, že v něm lze spouštět příkazy a programy stejně tak jako v jakémkoliv jiném shellu, ale také dokáže interaktivně vyhodnocovat různé operace s proměnnými, jednoduše tisknout výstup těchto operací na obrazovku a tak dále. Také jako například v Pythonu, Haskellu nebo mnoha dalších skriptovacích jazycích s interpretem, umožňujícím práci v interaktivním režimu, je možné jednoduše sečíst dvě proměnné a výsledek se okamžitě zobrazí na výstup.

```
PS> 1 + 2
3
```

PowerShell je rozšiřitelný – je tak možné psát vlastní cmdlety, funkce a skripty a vytvářet z nich moduly, které je možné distribuovat.

Kromě lokálního systému PowerShell umožňuje spravovat i systém vzdálený.

## 1.2 Představení Bashe

Bash je shell a programovací jazyk, jehož historie sahá až do roku 1989 [14]. V tomto roce vytvořil Brian Fox, zaměstnanec Free Software Foundation [15], Bash jako hlavní shell pro projekt GNU. Ten měl být náhradou za Bourne shell, který tehdy byl v Unixu verze 7. Bash je tedy reimplementací a vylepšením Bourne shellu a zachovává s ním z velké části zpětnou kompatibilitu.

Jelikož je Bash základní shell pro operační systémy postavených na projektu GNU, je na systémech GNU/Linux nejrozšířenější. Použitý je však i v operačním systému Mac OS X. Kombinuje v sobě výhody Korn shellu a C shellu a splňuje POSIXové standardy ohledně shellů (IEEE Standard 1003.1). Některé nověji přidané prvky však tyto standardy nesplňují [16].

Vzhledem k jeho stáří je jeho syntaxe a mentalita mírně neintuitivní pro lidi, kteří jsou zvyklí na programování v moderních programovacích či skriptovacích jazycích. Těmto jazykům se však svou funkčností bez problémů vyrovná. Problémem je, že k práci s tímto shellem je potřeba mít poměrně rozsáhlé znalosti, jinak člověk může používat například zbytečně složitou posloupnost příkazů namísto jednoho příkazu jednoduchého.

Aritmetické operace, které není problém provést v PowerShellu, není možné vykonat v Bashi tak jednoduchým způsobem. Stejný vstup jako v ukázce pro PowerShell skončí chybovým hlášením.

```
$ 1 + 2
-bash: 1: příkaz nenalezen
```

Tomu tak je z toho důvodu, že první řetězec oddělený mezerou je považován za název příkazu a další řetězce oddělené mezerami jsou považovány za argumenty příkazu. Pokud bychom chtěli dosáhnout stejného výsledku jako v PowerShellu, je potřeba poněkud složitější výraz, ve kterém je vyhodnocena aritmetická operace.

```
$ echo $((1 + 2))
3
```

## 1.3 Cmdlety

Cmdlety jsou jednoduché programy s konzolovým rozhraním, které jsou spouštěny v příkazové řádce PowerShellu. Každý cmdlet má nějakou specifickou funkci, například Get-Process slouží k výpisu procesů, které právě běží v počítači, Get-ChildItem vypíše obsah adresáře atd. Jednotlivé cmdlety je možné na sebe napojovat rourami a vytvářet tak složitější příkazy, stejně jako v Linuxu. Ostatně, roury se v tomto shellu používají podstatně častěji, než tomu bylo u shellu Command Prompt (cmd.exe) ve starších verzích operačních systémů Windows.

Pojmenování jednotlivých cmdletů je voleno systematicky tak, aby vypovídalo o jeho účelu. Název se skládá ze dvou částí – anglického slovesa pro danou operaci (verb) a názvu objektu (noun), na kterém je operace vykonávána. Za hlavní cmdlet se dá považovat Get-Command, který vrací přehled všech dostupných cmdletů, aliasů, funkcí a dalších příkazů. Lze mu předat různé parametry, podle kterých zobrazí výsledky. Například přepínačem -verb s parametrem „Get“ se vypíší pouze příkazy typu Get. Nejpoužívanější operace jsou v tabulce v příloze (Příloha A).

Systematické pojmenování cmdletů je velice výhodné, protože není problém příkazem Get-Command s patřičným parametrem získat seznam všech operací, které například formátují výpisy PowerShellu nebo pracují s určitým objektem. V tomto ohledu je Linux obecně ve velké nevýhodě, jelikož kvůli použití unixových utilit, které se jmenují každá zcela jinak, je potřeba mít rozsáhlejší znalosti pro práci se systémem nebo používat nějaký manuál, kde jsou jednotlivé příkazy popsány.

V PowerShellu lze vytvářet tzv. aliasy. To znamená, že příkaz lze používat pod jiným názvem. Toho je využito tím způsobem, že existují aliasy, které se jmenují jako unixové utility, ale ve skutečnosti jsou to cmdlety PowerShellu. Příklady budou uvedeny dále.

### 1.3.1 Nápověda

PowerShell disponuje příkazem Get-Help, který zobrazuje nápovědu pro jednotlivé cmdlety. Stručná nápověda pro jakýkoliv cmdlet se dá zobrazit přepínačem -?, ale příkazem Get-Help lze zobrazit i detailnější informace. Jedná se o obdobu manuálových stránek v Linuxu. A právě uživatele tohoto systému potěší fakt, že pro tento příkaz existuje alias man, takže příkaz pro použití nápovědy je poté stejný jako by byl v Linuxu (např. man Get-Command).

V základu nápověda obsahuje popis syntaxe, seznam parametrů a další spíše stručné informace. Pro získání kompletní nápovědy jsou dvě možnosti. Buďto lze použít přepínač -Online a získat tak nápovědu na webových stránkách nebo použít příkaz Update-Help (je potřeba spustit konzoli s právy administrátora (Obrázek 2)), který nahraje podrobné informace včetně ukázek použití daného příkazu. Poté lze využívat parametru -Detailed pro získání podrobných informací.

### 1.3.2 Základní cmdlety

V této podkapitole budou uvedeny nejpoužívanější cmdlety s příklady jejich použití. Představeny jsou zejména programy pro vstupně/výstupní operace. Nejdříve se budeme zabývat výstupem z PowerShellu na obrazovku a poté operacemi se soubory.

#### *1.3.2.1 Výstup na obrazovku*

Pokud chceme zobrazit nějaký text, můžeme použít dva cmdlety. Prvním je Write-Host. Ten slouží k výpisu přímo na obrazovku a neposílá svůj výstup do roury. Výstup na obrazovku můžeme různě vzhledově upravovat, například měnit barvy textu, pozadí, specifikovat čím oddělovat jednotlivé zobrazované objekty atd.

```
PS> Write-Host "Ahoj" -ForegroundColor Red
Ahoj
```

Druhý příkaz, Write-Output, není tak úplně pro výpis na obrazovku. Své vstupní objekty posílá na výstup do roury. Pokud však za tímto příkazem žádná roura není, výstup je vypsán na obrazovku a výsledek je stejný jako v případě Write-Host. Většinou tento cmdlet není potřeba, jelikož jakýkoliv příkaz na konci roury vypíše výstup na obrazovku i bez tohoto cmdletu. Zmiňuji ho zde však proto, že existuje alias echo, který je ekvivalentem ke stejnojmennému příkazu v Bashi. Ten také umožňuje poslat svůj výstup do roury. V PowerShellu tedy vlastně můžeme používat příkazy jako v Bashi (ale pouze některé). Takovýchto aliasů je nastaveno velké množství. Více jich je uvedeno v příloze (Příloha B).

```
PS> echo "Echo: Ahoj"
Echo: Ahoj
```

Příkaz Get-Alias zjistí, jaký je původní cmdlet s aliasem, který uvedeme jako argument -Name.

```
PS> Get-Alias -Name echo

CommandType     Name                    ModuleName
-----------     ----                    ----------
Alias           echo -> Write-Output
```

#### *1.3.2.2 Práce se soubory*

Pro přečtení souboru existuje v PowerShellu cmdlet Get-Content. Ten čte po řádcích obsah zadaného souboru a jeho výstupem je kolekce objektů (konkrétně textových řetězců) reprezentujících jednotlivé řádky. Nejdříve zapíšeme do souboru několik řádků:

```
PS> For($i = 0; $i -lt 5; $i++) {Write-Output "Řádek číslo $i" >>
text.txt}
```

Nyní si soubor načteme do proměnné:

```
PS> $soubor = Get-Content .\text.txt
PS> $soubor[0]
Řádek číslo 0
PS> $soubor[1]
Řádek číslo 1
```

Vidíme, že proměnná je pole, které obsahuje jednotlivé řádky souboru. Více k polím bude napsáno v samostatné kapitole.

## 1.4 Přesměrování vstupu/výstupu

Díky operátorům pro přesměrování vstupu nebo výstupů lze například jednoduše zapisovat do souborů nebo oddělit chybový výstup příkazu od standardního a každý tak zapisovat zvlášť. Můžeme tak například do jednoho souboru zapisovat normální výstup z programu a do druhého zapisovat pouze chyby [17].

### 1.4.1 Roury

Nedílnou součástí prakticky všech současných (a všech současných unixových) shellů jsou roury. Poprvé byly použity v Unixu v roce 1972 a slouží k napojení výstupu prvního programu na vstup druhého [18]. Tímto způsobem se dá na sebe bez problémů napojit několik programů, které často slouží jako tzv. filtry. To znamená, že tyto programy přijímají data na vstup, nějakým způsobem je upravují a poté je opět posílají na svůj výstup [19]. Znakem pro rouru je svislá čára |.

#### *1.4.1.1 PowerShell*

V PowerShellu roury pracují s objekty, jelikož výstupem a vstupem všech cmdletů jsou objekty [20]. Lze tak přes rouru poslat celý složitý objekt a vykonávat na něm různé operace [21], což poskytuje široké možnosti využití. Jednoduchým způsobem se tak dají provádět poměrně složité operace. To je velký rozdíl oproti Bashi. Zápis příkazů s rourou je lépe čitelný, než používání různých pomocných proměnných, které navíc zbytečně zabírají místo v paměti.

Na tomto místě by bylo vhodné zmínit cmdlet Where-Object, který slouží jako filtr. Na svůj výstup posílá pouze objekty přijaté ze svého standardního vstupu, které splňují podmínku uvedenou ve složených závorkách za příkazem. V ukázce je také použit příkaz Get-ChildItem, který slouží k získání obsahu kontejneru, kterým je v našem případě složka "C:\Windows". Tím tedy získáme objekty reprezentující soubory v této složce. Použití vypadá následovně:

```
PS> Get-ChildItem -Path "C:\Windows" | Where-Object {$_.Length -gt 1MB}


    Directory: C:\Windows


Mode        LastWriteTime          Length      Name
----        -------------          ------      ----
-a---       25.2.2011 7:19         2871808     explorer.exe
-a---       3.5.2013  1:54         2077530     WindowsUpdate.log
```

Tímto příkazem jsme vypsali všechny soubory obsažené v adresáři "C:\Windows", které splňují tu podmínku, že jsou větší než 1MB. Znak $_ reprezentuje objekt přenesený přes rouru. Porovnáváme tedy vlastnost objektu – velikost – s námi uvedenou hodnotou. Ta rozhodně stojí za povšimnutí. PowerShell totiž dokáže pracovat s jednotkami počtu bytů jako s číselnou hodnotou. Stejně tak dokáže pracovat s hexadecimálními čísly. Tyto speciální hodnoty jsou automaticky konvertovány do číselné hodnoty v desítkové soustavě. Lze je tedy používat k vyjádření hodnoty čísla.

```
PS> 1024 -gt 1KB
False
PS> 1025 -gt 1KB
True
PS> 0xFF -eq 255
True
```

#### *1.4.1.2 Bash*

Roury se v Bashi používají stejným způsobem jako v PowerShellu. Jediný, avšak velký rozdíl je v tom, že roury v Bashi jsou přizpůsobeny k přenosu informací ve formě textu. Alternativa k příkladu v PowerShellu by v Bashi nepoužívala roury, ale pouze jediný příkaz "find". Proto si ukážeme jak vypsat podadresář s největší velikostí.

```
$ du -d 1 | sort -n | tail -n 2 | head -n 1
31835   ./Lib
```

Příkaz du s parametrem "-d 1" vypíše velikosti pouze pro adresáře v současném umístění. Výstup je seřazen příkazem "sort -n", který seřadí řetězce na svém vstupu vzestupně a podle hodnoty velikostí adresářů a ne podle řetězce. Dále jsou přečteny poslední dva řádky obsahující adresáře s největší velikostí. Poslední řádek je však současný adresář, proto z těchto dvou řádků vypíšeme pouze první.

### 1.4.2 Přesměrování výstupu do souboru

Jak už název napovídá, přesměrováním výstupu do souboru lze docílit toho, že výstup z programu je zapsán do daného souboru. Za základní operátory přesměrování do souboru se dají považovat tyto:

**Tabulka 2 – Základní operátory přesměrování výstupu**

| Operátor | Význam |
|---|---|
| > | Přesměrování výstupu programu do souboru. Soubor je předtím vymazán. |
| >> | Připojení výstupu programu k obsahu souboru. |
| 2> | Přesměrování chybového výstupu programu do souboru. Soubor je předtím vymazán. |
| 2>> | Připojení chybového výstupu programu k obsahu souboru. |
| 2>&1 | Přesměrování chybového výstupu do standardního výstupu. Oba výstupy jsou tak spojeny. |

Výše uvedené operátory jsou společné jak pro PowerShell, tak pro Bash. Význam mají v obou jazycích totožný.

#### *1.4.2.1 PowerShell*

Ve verzi PowerShell 3.0 přibyla řada nových operátorů pro přesměrování varování, podrobného výstupu a ladících informací [17].

Přesměrování se používá takto:

```
PS> echo "Výstup je přesměrován" > soubor.txt
PS> Get-Content .\soubor.txt
Výstup je přesměrován

PS> echo "Výstup byl přesměrován podruhé" >> soubor.txt
PS> Get-Content .\soubor.txt
Výstup je přesměrován
Výstup byl přesměrován podruhé
```

##### 1.4.2.1.1 Out-File

Výstup do souboru skrze přesměrování je v PowerShellu v kódování Unicode. Pokud je potřeba do souboru zapsat text v jiném kódování, je třeba použít cmdlet Out-File. Ten má mnoho parametrů, jimiž může měnit své chování, a dokáže i přidávat text na konec souboru. Obyčejné zapsání do souboru potom vypadá takto:

```
PS> echo "Výstup je přesměrován přes Out-File" | Out-File soubor.txt
PS> Get-Content .\soubor.txt
Výstup je přesměrován přes Out-File
```

Připojíme text k původnímu souboru:

```
PS> echo "Připojeno přes Out-File " | Out-File -Append soubor.txt
PS> Get-Content .\soubor.txt
Výstup je přesměrován přes Out-File
Připojeno přes Out-File
```

Zapíšeme do souboru s kódováním UTF7 a přečteme v kódování Ascii:

```
PS> echo "čeština s háčky" | Out-File -Encoding utf7 soubor.txt
PS> Get-Content -Encoding Ascii .\soubor.txt
+AQ0-e+AWE-tina s h+AOEBDQ-ky
```

#### *1.4.2.2 Bash*

Jak bylo zmíněno výše, operátory pro přesměrování do souboru se používají v Bashi naprosto stejně jako v PowerShellu. Proto stačí uvést jeden příklad:

```
$ echo "První řádek" > soubor.txt
$ echo "Druhý řádek" >> soubor.txt
$ cat soubor.txt
První řádek
Druhý řádek
```

### 1.4.3 Přesměrování vstupu ze souboru

V PowerShellu neexistuje přesměrování na standardní vstup ze souboru. Cmdlety ani nejsou stavěny na to, aby přijímaly na standardní vstup text.

## 1.5 Proměnné

Proměnné slouží k dočasnému uchování informací a jsou naprosto nezbytné pro jakýkoliv skriptovací nebo programovací jazyk. V jazyce PowerShell jsou proměnné označeny symbolem $ (například $promenna), stejně jako v jazyce PHP. Tento znak se před proměnnou uvádí vždy, ať už jde o definici nové proměnné nebo její použití, na rozdíl od jazyku Bash, kde slouží pouze k získání hodnoty proměnné.

Pro výpis proměnné na standardní výstup stačí v PowerShellu uvést název proměnné ($promenna) nebo tento název uvést ve dvojitých uvozovkách (jednoduché uvozovky by nevypsaly obsah proměnné, ale jejich skutečný obsah – text „$promenna“). To je opět velice podobné jako v jazyce PHP. V jazyce Bash je pro výpis proměnné potřeba použít příkaz echo.

PowerShell využívá dynamické typové kontroly. To znamená, že typ obsahu proměnné se určuje až za běhu skriptu/programu a ne při kompilaci (protože žádná kompilace ani není). Typ obsahu proměnné lze také za běhu měnit. Lze tak například provést přiřazení nejdříve číselné hodnoty a poté řetězce. Pokaždé se přitom změní i samotný typ proměnné.

```
PS> $x = 10
PS> $x
10
PS> $x = "Ahoj"
PS> $x
Ahoj
```

Tomuto chování se však dá zabránit explicitním nastavení typu proměnné v hranatých závorkách.

```
PS> [int]$x = 10
PS> $x = "Ahoj"
Cannot convert value "Hoj" to type "System.Int32". Error: "Vstupní
řetězec nemá správný formát."
At line:1 char:1
+ $x = "Ahoj"
+ ~~~~~~~~~~
    + CategoryInfo : MetadataError: (:) [],
ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

V tomto případě již nelze změnit obsah proměnné na jiný typ, pokud nově přiřazovaná hodnota nejde zkonvertovat na odpovídající typ proměnné. Proměnnou však lze opět přetypovat (prosté určení jiného typu), v tomto případě na řetězec:

```
PS> [String]$x = "Ahoj"
```

Dokonce lze přiřadit i hodnotu jiného typu. Ta je ale zkonvertována na odpovídající typ proměnné.

```
PS> $x = 10
PS> $x
10
PS> $x.GetType().FullName
System.String
```

### 1.5.1 Uživatelské proměnné

Práce s proměnnými je v PowerShellu i v Bashi velice podobná. Proměnné není potřeba nikde deklarovat, ale pro vytvoření nové proměnné jí stačí přiřadit požadovanou hodnotu. Názvy proměnných nejsou v PowerShellu case sensitive, na rozdíl od Bashe, a také v názvech proměnných se mohou vyskytovat jakékoliv znaky. V jistých případech je však potřeba název proměnné uvést ve složených závorkách ( {} ), zejména v případě, že by v názvu proměnné byly mezery nebo uvozovky.

Při přiřazování hodnoty do proměnné může být znaménko rovnosti oddělené mezerou jak od proměnné, tak od přiřazované hodnoty. To v jazyce Bash není možné a mezi proměnnou, operátorem a operandem se nesmí vyskytovat mezery. Výhodou PowerShellu oproti Bashi je, že přiřazování lze zřetězit a tak uložit jednu hodnotu do více proměnných.

```
PS> $x = $y = 10
```

Toto v Bashi nejde, jelikož se vyhodnotí pouze první přiřazení a zbytek se považuje za řetězec, který se tak uloží do první proměnné.

```
$ x=y=10

$ echo $x
y=10
```

Díky tomu, že operátor přiřazení dokáže pracovat s poli mírně neobvyklým způsobem, lze například zjednodušit postup pro výměnu dvou proměnných nebo inicializovat více proměnných různými hodnotami na jednom řádku.

```
# Výměna hodnot dvou proměnných:
PS> $x = 1
PS> $y = 2
PS> $x, $y = $y, $x
PS> $x
2
PS> $y
1
```

```
# Inicializace více proměnných různými hodnotami:
PS> $x, $y = 11, 22
PS> $x
11
PS> $y
22
```

Práce s poli bude zmíněna v samostatné podkapitole.

PowerShell uchovává záznam o všech proměnných ve virtuálním disku s názvem variable:. To znamená, že pro výpis všech používaných proměnných lze použít příkaz dir.

```
PS> Dir variable:
```

V Bashi lze pro výpis všech proměnných použít příkaz compgen nebo set.

```
# Příkaz compgen vypíše názvy nastavených proměnných
$ compgen -v

# Příkaz set vypíše názvy proměnných i s jejich hodnotami
$ set
```

Všechny proměnné, které jsou vytvořené uživatelem, jsou při uzavření konzole vymazány.

PowerShell nabízí rozsáhlé možnosti při práci s proměnnými, například možnost vytvářet konstanty, mazat proměnné a dokonce k proměnným přidávat popisky. Pro výpis všech operací, které lze provést s proměnnými, lze použít již dříve zmíněný příkaz Get-Command.

```
PS> Get-Command -Noun Variable

CommandType     Name              ModuleName
-----------     ----              ----------
Cmdlet          Clear-Variable    Microsoft.PowerShell.Utility
Cmdlet          Get-Variable      Microsoft.PowerShell.Utility
Cmdlet          New-Variable      Microsoft.PowerShell.Utility
Cmdlet          Remove-Variable   Microsoft.PowerShell.Utility
Cmdlet          Set-Variable      Microsoft.PowerShell.Utility
```

### 1.5.2 Proměnné prostředí

Kromě uživatelských proměnných existují také takzvané proměnné prostředí. To jsou proměnné, které jsou přístupné v každé relaci jak u PowerShellu, tak u Bashe a jsou také přístupné i z jiných programů. Tyto proměnné obsahují důležité informace o operačním systému, jako je například adresář s nainstalovaným operačním systémem, domovský adresář uživatele, název operačního systému nebo seznam důvěryhodných adresářů (PATH).

#### *1.5.2.1 PowerShell*

S proměnnými prostředí se pracuje stejně jako s běžnými proměnnými s tím rozdílem, že jakékoliv jejich změny jsou pouze dočasné v rámci relace, jelikož se pracuje pouze s lokální kopií skutečných proměnných prostředí. To znamená, že když je konzole zavřena, změny jsou ztraceny. Pro skutečnou změnu systémových proměnných se používají metody .NET.

Všechny systémové proměnné jsou ve virtuálním disku env:. To znamená, že příkazem

```
PS> dir env:
```

lze vypsat všechny systémové proměnné.

Výpis obsahu systémové proměnné s cestou, kam se defaultně instalují programy, se provede takto:

```
PS> $env:ProgramFiles

C:\Program Files
```

Smazání té samé proměnné prostředí je stejné jako smazání souboru na disku. Provede se tímto příkazem:

```
PS> del env:\ProgramFiles
```

#### 1.5.2.2 Bash

V Bashi jsou systémové proměnné přístupné stejně jako běžné proměnné přímo z konzole nebo skriptu. Je také konvencí, že názvy proměnných prostředí jsou psané velkými písmeny. Jakékoliv změny těchto proměnných jsou ztraceny s uzavřením konzole. Pro trvalou změnu některé z proměnných prostředí se používá soubor .bashrc, který slouží pro nastavení konzole. Každý uživatel systému má vlastní soubor – není společný pro všechny.

Pro výpis všech proměnných prostředí lze použít příkaz printenv.

```
$ printenv
HOSTNAME=Taliesin
SHELL=/bin/bash
USER=Jirka
LANG=cs_CZ.UTF-8
…
```

Pro výpis konkrétní proměnné stačí použít příkaz echo:

```
$ echo $HOSTNAME
Taliesin
```

## 1.6 Objekty

PowerShell má jednu ohromnou zvláštnost v porovnání s jakýmikoliv shelly jak z Linuxu, tak z jiných operačních systémů. Používá objektově orientovaný přístup. Všechno v PowerShellu je objektem. I výstupy z jednotlivých příkazů jsou objekty, které jsou až při výstupu do konzole převedeny na čitelný text. Právě to umožňuje formátování výstupu z příkazů díky použití rour a formátovacích cmdletů. Roury totiž dokážou předávat objekty mezi jednotlivými cmdlety. Mezi zmíněné formátovací příkazy se řadí například Format-Table, který formátuje výstup do tabulky, nebo Format-List, který výstup formátuje do seznamu.

Jak už bylo zmíněno dříve, PowerShell je postaven na frameworku .NET a tak všechny objekty mají původ právě v objektech z .NET. Dokonce jde získávat objekty přímo z tohoto frameworku a pracovat s nimi, a pokud by toto nestačilo, jde volat i Windows API nebo používat COM komponenty.

V Bashi žádné objekty neexistují. Hlavní mentalitou Unixu, ze kterého Bash pochází, je, že všechny programy přijímají za vstup člověkem čitelný text a výstup z nich je také text.

### 1.6.1 Vytvoření objektu

Vytvoření nového objektu lze docílit použitím příkazu New-Object. Tento příkaz vytvoří novou instanci objektu typu COM nebo .NET v závislosti na použitém přepínači (bez přepínače defaultně vytváří objekty .NET). Jako parametr se zadá název .NET třídy. K vytvoření prázdného objektu lze využít třídy Object.

```
PS> $mujObjekt = New-Object Object
```

Proměnná $mujObjekt nyní obsahuje objekt, který nemá žádné atributy a obsahuje pouze čtyři základní metody, které jsou ve třídě Object. O tom se můžeme přesvědčit použitím příkazu Get-Member, který slouží k získání informací o členech (atributech a metodách) a typu objektu.

```
PS> $mujObjekt | Get-Member

      TypeName: System.Object

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

### 1.6.2 Práce s atributy

Předchozím příkazem New-Object jsme získali prakticky prázdný objekt, který nemá žádné vlastnosti a ani na něm nelze provádět žádné operace. Vlastnosti a metody, které jsou tak trochu „navíc“, se dají přidat příkazem Add-Member. Do objektu $mujObjekt přidáme atribut se jménem (definováno parametrem -Name) Velikost a s hodnotou 100 (hodnota definována parametrem -Value). Typ atributu (-MemberType) nastavíme na NoteProperty, což je atribut pocházející z prostředí PowerShellu (není z prostředí .NET), který je definován jako klíč=hodnota (v tomto případě Velikost=100).

```
PS> Add-Member -InputObject $mujObjekt -MemberType NoteProperty -Name
Velikost -Value 100
```

Nyní můžeme zobrazit hodnotu atributu tímto příkazem:

```
PS> $mujObjekt.Velikost
100
```

Pokud na obrazovku vypíšeme proměnnou obsahující objekt, zobrazí se atributy objektu.

```
PS> $mujObjekt

Velikost
--------
100
```

Jako atribut objektu může být samozřejmě i jiný objekt.

### 1.6.3 Práce s metodami

Metody se přidávají stejně tak jako atributy použitím příkazu Add-Member za použití parametru -MemberType nastaveného na hodnotu ScriptMethod. To znamená, že je přidávána metoda, jejíž tělo je skript PowerShellu. Parametrem -Name definujeme název metody a parametrem -Value napíšeme blok skriptu ve složených závorkách, jehož výsledná hodnota bude vrácena metodou.

```
PS> Add-Member -InputObject $mujObjekt -MemberType ScriptMethod -Name
Pozdrav -Value {echo "Ahoj, světe!"}
```

Nyní můžeme na objektu $mujObjekt zavolat metodu Pozdrav.

```
PS> $mujObjekt.Pozdrav()
Ahoj, světe!
```

Pokud bychom uvedli metodu bez závorek, došlo by k vypsání informací o metodě.

```
PS> $mujObjekt.Pozdrav

Script                  : echo "Ahoj, světe!"
OverloadDefinitions     : {System.Object Pozdrav();}
MemberType              : ScriptMethod
TypeNameOfValue         : System.Object
Value                   : System.Object Pozdrav();
Name                    : Pozdrav
IsInstance              : True
```

Nyní můžeme vypsat všechny členy objektu příkazem Get-Member.

```
PS> $mujObjekt | Get-Member

      TypeName: System.Object

Name              MemberType        Definition
----              ----------        ----------
Equals            Method            bool Equals(System.Object obj)
GetHashCode       Method            int GetHashCode()
GetType           Method            type GetType()
ToString          Method            string ToString()
Velikost          NoteProperty      System.Int32 Velikost=100
Pozdrav           ScriptMethod      System.Object Pozdrav();
```

### 1.6.4 Výpis atributů objektu – formátovací cmdlety

Vypsat atributy objektů a jejich hodnoty je velice jednoduché díky formátovacím cmdletům. Ty přijímají objekty přes rouru nebo přes parametr -InputObject a poté je určitým způsobem zobrazují. K tomuto účelu existuje hned několik formátovacích cmdletů, které se liší tím, v jakém formátu jsou atributy a hodnoty zobrazeny. Atributy, které chceme zobrazit, se uvádějí za formátovací cmdlet oddělené čárkami. Pokud chceme vypsat všechny atributy, lze za formátovací příkaz uvést *. Zde si uvedeme hlavní dva formátovací cmdlety.

#### *1.6.4.1 Format-List*

Tento příkaz zobrazí atributy objektu jako list ve formátu atribut : hodnota:

```
PS> dir C:\Windows\System32\notepad.exe | Format-List Name, Length,
Directory


Name        : notepad.exe
Length      : 193536
Directory   : C:\Windows\System32
```

### 1.6.4.2 Format-Table

Tento cmdlet zobrazí atributy vybraného objektu jako tabulku. Tento způsob zobrazení se hodí zejména pro zobrazení údajů o více objektech najednou.

```
PS> C:\Users\Jirka> dir C:\Windows\*.ini | Format-Table Name, Length,
Directory

Name                Length      Directory
----                ------      ---------
game.ini            272         C:\Windows
HotFixList.ini      1035        C:\Windows
msdfmap.ini         1405        C:\Windows
MSYS.INI            59          C:\Windows
ODBC.INI            384         C:\Windows
system.ini          219         C:\Windows
win.ini             403         C:\Windows
```

## 1.7 Pole

Pole jsou v PowerShellu velice často využívána. Například když je výstupem příkazu více hodnot a jsou uloženy do proměnné, bude tato proměnná běžným indexovým polem obsahujícím výstupní hodnoty.

### 1.7.1 Indexová pole

Indexová pole jsou pole, kde jsou prvky jednoznačně určeny podle jejich čísla indexu, které je zpravidla celé číslo. Při vkládání prvku nebo čtení prvku se na dané místo vkládání/čtení odkazujeme právě pomocí tohoto indexu. Prvky pole jsou v paměti počítače většinou ukládány za sebe podle hodnoty indexu na přesně určená místa. Proto bývá provádění operací s těmito poli velice rychlé.

#### *1.7.1.1 Vytvoření pole*

V PowerShellu můžeme pole získat několika způsoby. Prázdné pole se dá vytvořit operátorem přiřazení tím způsobem, že na pravé straně operátoru zadáme více hodnot oddělených čárkami.

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

Rovněž lze použít operátor rozsahu (..), který určuje, v jakém rozsahu se mají prvky do pole přidat. Vytvoření pole celých čísel v rozsahu od 1 včetně do 10 včetně se provede takto:

```
PS> $a = 1..10
```

Pole v PowerShellu je defaultně nastaveno tak, že nemusí být homogenní. Pokud se vysloveně neurčí, že pole může obsahovat pouze prvky určité třídy, předpokládá se, že prvky pole mohou být všechny instance třídy Objekt. Jelikož je všechno v PowerShellu objekt (a vychází ze třídy Object), znamená to, že lze do takového pole vkládat prvky různých tříd. Například:

```
PS> $a = "V poli jsou ", 3, " retezce a ", 2, " promenne"
```

Pro vytvoření pole se silným typováním (v poli můžou být pouze hodnoty jednoho typu) je třeba před pole uvést typ pole (typ prvků, které mohou být v poli uloženy) operátorem hranatých závorek. Například typ pole celých 32-bitových čísel:

```
PS> [int32[]]$sa = 1,2,3
```

```
PS> $sa
1
2
3
PS> [int32[]]$sa = 1,2,3,"A"
Cannot convert value "A" to type "System.Int32". Error: "Vstupní řetězec
nemá správný formát."
…
```

Pro vytvoření pole jde také použít jazyková konstrukce @(…), díky které lze vytvořit pole hodnot, které jsou v ní obsaženy nebo případně prázdné pole, pokud jsou závorky prázdné. V této konstrukci se mohou vyskytovat i příkazy, z jejichž návratové hodnoty bude pole vytvořeno.

```
# Vytvoření prázdného pole
PS> $a = @()

# Vytvoření pole s hodnotami 1,2 a 3
PS> $a = @(1,2,3)
```

#### 1.7.1.2 Práce s polem

Prvky v poli se adresují způsobem, jaký známe z mnoha jiných programovacích či skriptovacích jazyků. Za proměnnou pole se uvede operátor hranatých závorek obsahující index prvku, ke kterému chceme přistupovat. Pole začíná vždy indexem 0. Při zadání záporných hodnot indexu se zpřístupňují prvky od posledního k prvnímu, tedy -1 pro poslední prvek, -2 pro předposlední atd.

```
PS> $a = 1,2,3
PS> $a[2]
3
PS> $a[-1]
3
```

Z jednoho pole lze vybrat i více prvků najednou, stačí zadat jejich indexy oddělené čárkami do hranatých závorek za proměnnou pole. Výsledkem výběru je opět pole.

```
PS> $a = 1..5
PS> $a
1
2
3
4
5
PS> $a[0,1,4]
1
2
5
```

Přidávání do pole se provede jednoduše operátorem +=. Velikost pole se nedá změnit, proto tento operátor provede vytvoření nového, většího pole, zkopírování starých prvků, přidání nového prvku a navrácení nového pole zpět do proměnné.

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

Menší problém nastává při kopírování pole do jiné proměnné. Nestačí pouze tento příkaz:

```
PS> $a = $b
```

Došlo by totiž ke zkopírování reference na pole. Tím by proměnné $a i $b ukazovaly na jedno stejné pole a změny pole $a by se promítly i do pole $b. Je tak potřeba použít metody Clone, která vytvoří kopii pole.

```
PS> $a = $b.Clone()
```

#### *1.7.1.3 Bash*

V novějších verzích jazyku Bash jsou pole podporována. Jedná se však pouze o jednorozměrná pole [22]. V původní verzi Bashe pole ani neexistovala. Indexová pole byla přidána až ve verzi 2 [23] (rok 1996 [24]) a ve verzi 4 (rok 2009) přibyla dokonce pole asociativní [25].

Indexová pole jsou navíc tzv. řídká pole, což znamená, že indexy prvků v poli nemusí jít po sobě, ale může existovat prvek s indexem 1 a ihned po něm prvek s indexem 5.

Vytvoření pole je v jazyce Bash stejně jednoduché jako v PowerShellu, avšak provádí se jiným způsobem. Lze buďto vytvořit pole výčtem jednotlivých prvků v kulatých závorkách:

```
$ pole=( 10 20 30 40 )
```

nebo přiřazením hodnot přímo konkrétnímu indexu v poli:

```
$ pole[1]=jedna
$ pole[15]=patnáct

# vypsání všech prvků pole – vysvětleno dále
$ echo ${pole[*]}
10 jedna 30 40 patnáct
```

Také lze zkombinovat oba postupy:

```
$ pole2=( 10 20 [15]=1515 )
```

Pro práci s poli se používá několik speciálních jazykových konstrukcí. Například pro získání hodnoty z pole na daném indexu nestačí použít proměnnou takto:

```
$pole[index]
```

jak by tomu bylo v PowerShellu, ale musí se použít složených závorek:

```
${pole[index]}
```

Pro zjednodušení práce s polem lze také používat jisté „triky“:

```
${pole[*]}        # Vrací všechny prvky v poli
${!pole[*]}       # Vrací všechny indexy v poli
${#pole[*]}       # Vrací počet prvků v poli
${#pole[0]}       # Vrací délku prvku s indexem 0
```

```
# ukázka příkazů

$ echo ${pole2[*]}
10 20 1515

$ echo ${!pole2[*]}
0 1 15

$ echo ${#pole2[*]}
3

$ echo ${#pole2[0]}
2
```

Namísto znaku * lze také používat znak @. Rozdíl mezi nimi je znát, pokud jsou proměnné pro vrácení všech prvků v poli použity uvnitř uvozovek. To je podrobněji vysvětleno v kapitole o funkcích.

### 1.7.2 Hashovací tabulky (asociativní pole)

Hashovací tabulky jsou datové struktury, které jsou nejčastěji používané pro implementaci asociativního pole. Pro zpřístupnění uložené hodnoty používají klíč na rozdíl od pole, které používá číselný index. V případě PowerShellu je klíčem nejčastěji řetězec. Práce s hashovacími tabulkami je až na pár výjimek stejná jako práce s poli.

#### 1.7.2.1 Vytvoření hashovací tabulky

Pro vytvoření hashovací tabulky se používá operátor @{}, což je ekvivalentní k operátoru @() u pole. Rozdíl je, že vkládané prvky se uvádí ve tvaru klíč=hodnota a tyto jednotlivé prvky se oddělují středníky.

```
PS> $hashTab = @{"ID1"="John Doe"; "ID2"="Jane Doe"}
PS> $hashTab

Name                           Value
----                           -----
ID1                            John Doe
ID2                            Jane Doe
```

#### 1.7.2.2 Práce s hashovacími tabulkami

Výpis hodnoty spojené s určitým klíčem lze provést dvěma způsoby. První je stejný jako u pole – příslušný klíč se uvede do hranatých závorek za proměnnou hashovací tabulky.

```
PS> $hashTab["ID1"]
John Doe
```

Druhý způsob je zápis klíče za proměnnou přes tečku, jelikož každý klíč je i atributem hashovací tabulky.

```
PS> $hashTab.ID2
Jane Doe
```

Všechny klíče v hashovací tabulce lze získat z atributu keys.

```
PS> $hashTab.Keys
ID1
ID2
```

Přidání nových nebo změna již uložených hodnot lze opět provést dvěma způsoby podobnými jako u výpisu. První způsob je přidání/změna přes hranaté závorky a druhý přes atribut.

```
PS> $hashTab["ID3"] = "Jack Black"
PS> $hashTab.ID1 = "Black Jack"
PS> $hashTab

Name                           Value
----                           -----
ID1                            Black Jack
ID3                            Jack Black
ID2                            Jane Doe
```

Odebrání hodnoty i s klíčem se provede metodou Remove.

```
PS> $hashTab.remove("ID3")
PS> $hashTab

Name                           Value
----                           -----
ID1                            Black Jack
ID2                            Jane Doe
```

Při kopírování hashovací tabulky vyvstává stejný problém jako při kopírování pole. Problém i jeho řešení jsou uvedeny v předchozí podkapitole Indexová Pole.

#### *1.7.2.3 Bash*

V jazyce Bash byly přidány asociativní pole poměrně nedávno (viz kapitolu Indexová Pole). Pracuje se s nimi naprosto stejně jako s indexovými poli, až na to, že se místo indexů používají klíče a nejprve se musí deklarovat.

Deklarace asociativního pole se provede příkazem declare.

```
$ declare -A asocPole
```

Poté již lze s asociativním polem normálně pracovat.

```
$ asocPole["ID1"]="John Doe"
$ asocPole["ID2"]="Jane Doe"
```

Lze také pole deklarovat a rovnou inicializovat:

```
$ declare -A asocPole2=( ["ID1"]="John Doe" ["ID2"]="Jane Doe" )
```

Příkazy pro práci s asociativním polem jsou víceméně stejné jako s normálním polem.

## 1.8 Funkce

Funkce je obecně v programovacích jazycích posloupnost instrukcí v pojmenovaném bloku, která vykonává specifickou úlohu a je samostatnou jednotkou, která může být v samotném programu použita na více různých místech. V PowerShellu tomu není jinak. Také je každá funkce v PowerShellu příkazem. Proto se s nimi pracuje stejně jako s cmdlety a nepoužívají se při jejím volání žádné kulaté závorky, jak je tomu například při volání funkce v jazyku C.

Klíčové slovo pro definování funkce je Function. V tomto ohledu je PowerShell stejný jako Bash i z hlediska syntaxe definice funkce.

Definice jednoduché funkce vypadá následovně.

```
Function pozdrav {
      echo "Ahoj"
}
```

Výhodou PowerShellu je, že funkce lze definovat i v interaktivní konzoli. Stačí výše uvedenou funkci vkládat řádek po řádku nebo celou zkopírovat a vložit do konzole. Poté stačí pouze stisknout enter, čímž se ukončí definování funkce v interaktivním režimu. Tato funkce po svém zavolání vypíše na obrazovku „Ahoj“.

```
PS> pozdrav
Ahoj
```

Často je však potřeba funkce, která neprovádí stále přesně tu samou úlohu, ale je potřeba její běh nějak ovlivnit zvenčí. K tomu slouží argumenty funkce. Ty lze funkci předávat stejným způsobem jako běžnému cmdletu. Konkrétní způsob předávání argumentů záleží na definici funkce. Funkci s argumenty lze implementovat několika různými způsoby. První způsob je neuvádět žádné argumenty, stejně tak jako v předchozím příkladu. V tom případě jsou všechny argumenty uloženy v poli s názvem $args. [26] [27]

```
Function pozdravUzivatele {
      echo "Ahoj, $($args -join ", ")"
}
```

Pokud použijeme tuto funkci a jako argumenty jí předáme jména, jsou tato jména uložena do pole $args a následně vypsána v textovém řetězci na obrazovku.

```
PS> pozdravUzivatele Jakub Jan Ryba
Ahoj, Jakub, Jan, Ryba
```

Další způsob, jak specifikovat parametry funkce, je zapsat parametry do kulatých závorek za název funkce při její definici.

```
Function pozdravUzivatele ( $jmeno ) {
      echo "Ahoj, $jmeno"
      echo "Nepozdravení uživatelé: $($args -join ", ")"
}
```

V tomto případě je do parametru $jmeno uložen pouze první argument uvedený za názvem funkce. Zbylé argumenty jsou opět uloženy v poli $args, jak je vidět při volání této funkce s více argumenty.

```
PS> pozdravUzivatele Jakub Jan Ryba
Ahoj, Jakub
Nepozdravení uživatelé: Jan, Ryba
```

Pokud bychom chtěli, aby byla zadaná hodnota konkrétního typu (číslo, řetězec atd.), je potřeba před parametr zadat název příslušného datového typu do hranatých závorek. Pokud bychom navíc chtěli, aby byla nastavena hodnota, pokud není funkci předán žádný argument, uvede se tato hodnota za parametr přes znaménko přiřazení.

```
Function secti ( [int]$x="1", [int]$y="1" ) {
      echo "$($x+$y)"
}
```

Nyní lze zadat dva argumenty, které budou sečteny. Pokud zadáme pouze jeden argument, přičte se k jeho hodnotě číslo 1 a výsledná hodnota se zobrazí. Také pokud zadáme argument jiného typu než celé číslo nebo nějaký argument, který lze převést na číselnou hodnotu (například řetězec obsahující číslo), funkce skončí chybou. V tomto případě byla schválně použita jiná funkce než v předchozích ukázkách. Pokud by byla použita předchozí funkce a jako typ byl uveden řetězec, nastal by problém, jelikož prakticky vše lze na řetězec převést.

```
# Sečtení dvou čísel.
PS> secti 5 5
10

# Přičtení defaultní hodnoty 1 k zadanému číslu.
PS> secti 2
3

# Převod řetězce na číslo.
PS> secti "10" 10
20
```

```
# Chyba - řetězec "Ahoj" není možné převést na celé číslo.
PS> secti "Ahoj"
secti : Cannot process argument transformation on parameter 'x'. Cannot
convert value "Ahoj" to type "System.Int32". Error: "Vstupní řetězec nemá
správný formát."
At line:1 char:7
+ secti "Ahoj"
+       ~~~~~~
    + CategoryInfo          : InvalidData: (:) [secti],
ParameterBindingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,secti
```

V této funkci jsou parametry načítány podle pořadí argumentů. Naštěstí je funkci secti jedno, v jakém pořadí jsou argumenty zadány, jelikož je sčítání komutativní. Avšak lepší, než v takovýchto případech definovat uživateli neznámé defaultní hodnoty parametrů, je zakázat vykonání funkce, pokud jeden z argumentů chybí. V tom případě lze vyvolat vlastní výjimku.

```
Function vydelDveCisla
(
      [int]$delenec="1",
      [int]$delitel=$(Throw "Je třeba zadat dělitele!")
)
{
      echo "$($delenec/$delitel)"
}
```

Ukázka chování funkce:

```
# Funkce je provedena bez problémů.
PS> vydelDveCisla 8 4
2

# Při chybějícím druhém argumentu je vyvolána výjimka.
PS> vydelDveCisla 8
Je třeba zadat dělitele!
At line:1 char:61
+ Function vydelDveCisla ( [int]$delenec="1", [int]$delitel=$(Throw "Je
třeba zada ...
```

Také lze použít pojmenované parametry, pokud chceme docílit toho, aby nebylo potřeba předávat argumenty pouze podle pořadí. Poté stačí při volání funkce jednoduše zadat název parametru před argument. Tohoto chování docílíme použitím bloku Param [28].

```
Function vydelDveCisla {
      Param ( [int]$delenec = 1,

              [Parameter(Mandatory=$True)]
              [int]$delitel )
      echo "$($delenec/$delitel)"
}
```

Tímto jsme provedli hned několik věcí. Za prvé, pokud zadáme pouze jeden argument, použije se přiřazení do parametru podle pozice. Takže je definována hodnota parametru $delenec, ale ne $delitel. Ale jelikož máme před parametrem $delitel nastaveno, že tento parametr je povinný, PowerShell požádá o dodatečné doplnění parametru.

```
PS> vydelDveCisla 10
cmdlet vydelDveCisla at command pipeline position 1
Supply values for the following parameters:
delitel: 2
5
```

Po dodatečném zadání hodnoty 2 proběhlo vydělení čísla 10 číslem 2 a výsledek je zobrazen.

Teď můžeme navíc předat pouze argument pro dělitele, jelikož jsou parametry pojmenované.

```
PS> vydelDveCisla -delitel 2
0.5
```

Defaultní hodnota dělence, číslo jedna, je vydělena číslem dvě, které jsme zadali jako argument. Pojmenované parametry tedy značně ulehčují používání funkcí, jelikož uživatel nemusí hlídat, zda jsou argumenty ve správném pořadí, ale může jejich pořadí libovolně měnit.

Zatím všechny uvedené funkce pouze vypisovaly text na výstup. To ověříme jednoduchým pokusem.

```
PS> $podil = vydelDveCisla 5 2
PS> $podil
2.5


PS> $podil.GetType()

IsPublic    IsSerial    Name        BaseType
--------    --------    ----        --------
True        True        String      System.Object
```

Z výsledků těchto příkazů lze vyčíst, že funkce vrací řetězcovou hodnotu typu String. Co když ale chceme z funkce vrátit nějakou vypočtenou číselnou hodnotu k dalšímu zpracování? K tomu slouží klíčové slovo Return.

```
Function secti ( [int]$x, [int]$y ) {
      Return $x + $y
}
```

Nyní funkce vrací součet dvou zadaných parametrů. To můžeme jednoduše ověřit.

```
PS> $soucet = secti 2 3
PS> $soucet
5
PS> $soucet.GetType()

IsPublic    IsSerial    Name        BaseType
--------    --------    ----        --------
True        True        Int32       System.ValueType
```

Díky těmto příkazům jsme si ověřili, že výstupem funkce je celé číslo typu Int32.

### 1.8.1 Bash

Jak již bylo zmíněno u definice funkce PowerShellu, Bash má pro tento účel stejnou syntaxi. I přístup k funkcím je stejný. V Bashi se používá klíčové slovo function a funkce se také chovají jako normální příkazy. K ohraničení bloku příkazů se dokonce používají složené závorky. Funkce je však potřeba definovat ve skriptu. Z této jednoduché ukázky je vidět, že syntaxe pro analogickou funkci v Bashi k první ukázkové v PowerShellu je stejná.

```
#!/bin/bash
# Skript func_pozdrav.sh

function pozdrav {
      echo "Ahoj"
}
```

Tuto funkci lze bez problémů používat ve stejném skriptu, kde je definovaná. Pro to, abychom mohli používat výše uvedenou funkci v konzoli, je potřeba použít na skript s funkcí příkaz source. Ten přečte a vykoná všechny příkazy v rámci současné konzole. Tím se nahraje funkce do konzole a je možné ji používat.

```
$ source func_pozdrav.sh
```

Nyní můžeme zavolat funkci pozdrav. Výsledkem bude výpis řetězce „Ahoj“ na obrazovku.

```
$ pozdrav
Ahoj
```

### *1.8.1.1 Parametry*

Parametry lze ve funkci získávat dvěma způsoby. Buďto podle jejich pozice nebo všechny najednou.

#### 1.8.1.1.1 Poziční parametry

Pokud získáváme parametry podle jejich pozice, používá se proměnná $č, kde č je pořadí předaného argumentu, které chceme získat. Proměnná $0 vždy obsahuje název skriptu s cestou tak, jak byl spuštěn. Pokud však na skript použijeme příkaz source a funkci voláme z konzole, v parametru bude uložen řetězec „-bash“. Toto si ukážeme na skriptu param_0_s.sh.

```
#!/bin/bash
# Skript param_0_s.sh

function param_0 {
      echo "\$0: $0"
}
param_0
```

Nyní zkusíme tento skript spustit různými způsoby.

```
$ ./param_0_s.sh
$0: ./param_0_s.sh


$ /home/Jirka/param_0_s.sh
$0: /home/Jirka/param_0_s.sh

$ source param_0_s.sh
$0: -bash
```

Jak již bylo napsáno výše, příkaz source přečte a *vykoná* příkazy uvedené ve skriptu. Proto se vykoná i funkce uvedená na konci skriptu.

S parametry uvedenými za funkcí se pracuje stejným způsobem, avšak tyto parametry jsou číslovány od čísla jedna. Pro výpis prvních tří parametrů za funkcí mějme následující skript.

```
#!/bin/bash
# Skript params.sh

function params {
      echo "\$1: $1, \$2: $2, \$3: $3"
}
params číslo1 číslo2 číslo3
```

Skript spustíme. Na výstup jsou vypsány tři argumenty předané funkci.

```
$ ./params.sh
$1: číslo1, $2: číslo2, $3: číslo3
```

#### 1.8.1.1.2 List parametrů

Pro získání všech parametrů lze použít proměnné $* a $@. Ty mají stejný význam jako u pole včetně rozdílného chování pokud jsou uvnitř uvozovek. Pro výpis všech argumentů za funkcí si ukážeme jednoduchý skript. Argumenty získané tímto způsobem nezahrnují poziční argument $0.

```
#!/bin/bash
# Skript params_all.sh

function params_all {
      for param in $*; do
            echo "$param"
      done
}
params_all číslo1 číslo2 číslo3 číslo4
```

Spuštěním skriptu vypíšeme argumenty za funkcí.

```
$ ./params_all.sh
číslo1
číslo2
číslo3
číslo4
```

Abychom viděli rozdíl mezi proměnnými $* a $@, spustíme následující skript.

```
#!/bin/bash
# Skript param_rozdil.sh

function params_all2 {
      for parama in "$*"; do
            echo "\$*: $parama"
      done
      for paramat in "$@"; do
            echo "\$@: $paramat"
      done
}
params_all2 číslo1 číslo2 číslo3 číslo4
```

Po spuštění skriptu dojde k tomu, že pokud je v uvozovkách proměnná $*, všechny proměnné, které obsahuje, se sloučí do jednoho velkého řetězce. Naopak pokud je v uvozovkách proměnná $@, tyto v ní obsažené proměnné se „obalí“ uvozovkami každá zvlášť (což je ve většině případů zamýšlené chování). Proto bude výstup z předchozího skriptu následující:

```
$ ./param_rozdil.sh
$*: číslo1 číslo2 číslo3 číslo4
$@: číslo1
$@: číslo2
$@: číslo3
$@: číslo4
```

## 1.9 Podmínky

Podmínky jsou v každém programovacím jazyce používány pro řízení toku programu. Používají výrazy, které mohou být jednoznačně vyhodnoceny jako pravdivé (true) nebo nepravdivé (false), a podle jejich výsledné hodnoty rozhodují o dalším chování programu.

V Bashi je ale porovnávání hodnot značně složitější než v PowerShellu. Pro porovnávání číselných hodnot Bash používá stejné operátory jako PowerShell pro všechna porovnávání a pro porovnávání řetězců používá „běžné“ operátory, známé z jiných programovacích jazyků. Viz tabulku B v příloze. Bash používá pro vyhodnocení výrazu program test nebo hranaté závorky. Časem se objevily i nové způsoby pro práci s výrazy a tak se v současné době dají použít i zdvojené obyčejné závorky (pro složitější číselné výrazy) a zdvojené hranaté závorky (pro pokročilé vyhodnocení výrazů s řetězci a soubory a také jednodušší zápis složitějších výrazů). Navíc nepoužívá klasické hodnoty 0 pro false a nenulovou hodnotu pro true, ale přesně naopak. 0 zde znamená true a ostatní hodnoty false. Je to z toho důvodu, že programy v Unixu vrací hodnotu 0 při úspěchu a nenulovou hodnotu v případě neúspěchu. [29] [13]

### 1.9.1 Vyhodnocení podmínek v PowerShellu

PowerShell používá jiné operátory pro porovnávání hodnot, než které jsou běžné u klasických programovacích jazyků (u shellů však bývají normální). Nepoužívá operátory rovnosti (= nebo ==) větší (>), menší (<) a podobné, ale pro porovnání rovnosti používá operátor -eq, pro operátor větší operátor -gt, pro menší -lt a další. Seznam všech operátorů pro porovnání je v tabulce v příloze (Příloha C). Operátor = totiž slouží výhradně pro přiřazování hodnot, > slouží k přesměrování a < je vyhrazený a zatím se nepoužívá. V tomto ohledu je PowerShell velice blízký Bashi.

Otestování rovnosti dvou čísel se například provede tímto jednoduchým příkazem:

```
PS> 1 -eq 1
True
```

### 1.9.2 Vyhodnocení podmínek v Bashi

Vyhodnocování podmínek je však v Bashi poněkud složitější. Je potřeba použít program test nebo hranaté závorky, výsledek bude totožný. Proměnná $? uchovává návratovou hodnotu předchozího programu, což je v tomto případě výsledek porovnání dvou čísel [30].

```
$ test 1 -eq 1; echo $?
0

$ [ 1 -eq 1 ]; echo $?
0
```

Výsledek porovnání v ukázce je číslo 0, jelikož číslo 1 se rovná číslu 1 a výraz tak má hodnotu pravda. Výše zmíněné příkazy pro porovnání jsou ale považovány za zastaralé. Pokud není potřeba zachovávat zpětnou kompatibilitu s Bourne shellem, je doporučeno používat novější dvojité hranaté závorky pro vyhodnocení podmínek a dvojité kulaté závorky pro práci s aritmetickými výrazy [16]. Jejich použití je také mnohem jednodušší.

```
$ (( 1 == 1 )); echo $?
0

$ [[ "a" != "b" ]]; echo $?
0
```

### 1.9.3 Větvení If

Příkaz If se používá k vykonání nějakého kusu kódu, pokud je splněna podmínka, nebo k rozvětvení běhu programu na dvě části.

#### *1.9.3.1 PowerShell*

Základní syntaxe tohoto příkazu je stejná jako například v jazycích Java nebo C.

```
# Skript if.ps1

$x = 2
If ($x -eq 2) {
      # Vykonání kódu při splnění podmínky
} Else {
      # Vykonání kódu při nesplnění podmínky
}
```

PowerShell však v příkazu If používá i nepovinný blok ElseIf, který se vykoná, pokud není splněna hlavní podmínka za příkazem If, ale je splněna podmínka za příslušným příkazem ElseIf. Tento příkaz lze použít, kolikrát je třeba.

```
# Skript elseif.ps1

$x = 10
If ($x -eq 2) {
      # Vykoná se, pokud je $x rovno dvěma
} ElseIf ($x -gt 10) {
      # Vykoná se, pokud je $x větší než 10
} ElseIf ($x -eq 11) {
      # Vykoná se, pokud je $x rovno jedenácti
} Else {
      # Vykoná se, pokud žádný z předchozích výrazů nebyl pravdivý
}
```

V tomto větvení nikdy neproběhne blok pro $x rovné jedenácti, jelikož po již splněné podmínce ($x je větší než 10) jsou následující podmínky ignorovány.

#### *1.9.3.2 Bash*

Příkaz if je v Bashi mírně odlišný od příkazu if v PowerShellu. Prvním rozdílem je, že nepoužívá složené závorky pro vymezení bloku příkazů. Druhý rozdíl je, že za výrazem if se neuvádí podmínka, ale příkaz, jehož návratová hodnota je pak vyhodnocena. Nejjednodušší použití tohoto příkazu vypadá takto:

```
#!/bin/bash
# Skript if.sh

if grep "user" /etc/passwd
then
      echo "Uživatel user existuje!"
else
      echo "Uživatel user neexistuje!"
fi
```

Tento skript vypíše na obrazovku „`Uživatel user neexistuje!`“, jelikož v ukázkovém systému není uživatel user přítomen.

Pokud bychom chtěli použít příkaz if k vyhodnocení podmínky, museli bychom použít příkaz test nebo hranaté, resp. zdvojené hranaté nebo zdvojené kulaté závorky, jak již bylo zmíněno. Pro vyhodnocení, zda je proměnná x rovna dvěma, použijeme následující skript.

```
#!/bin/bash
# skript if_expr.sh

x=2
if (( $x == 2 )); then
      echo "x == 2"
else
      echo "x != 2"
fi
```

Skript spustíme.

```
$ ./if_expr.sh
x == 2
```

Po spuštění skriptu se na obrazovku vypsal řetězec „x == 2“, což znamená, že podmínka byla vyhodnocena jako pravdivá.

Analogicky k příkazu ElseIf v PowerShellu existuje v Bashi příkaz elif, který má stejný význam a používá se takto:

```
#!/bin/bash
# Skript elif.sh
x=11
if (( $x == 2 )); then
      echo "x == 2"
elif (( $x > 10 )); then
      echo "x > 10"
elif (( $x == 11 )); then
      echo "x == 11"
else
      echo "Neplatí"
fi
```

Tento skript vypíše do konzole řetězec "x > 10". Je tomu tak ze stejného důvodu, jako u PowerShellu – po nalezení první vyhovující podmínky se provede příslušný blok kódu a další podmínky se nevyhodnocují.

### 1.9.4 Větvení Switch

Toto větvení se používá, pokud chceme testovat jednu proměnnou na více možných hodnot a pro každou tuto možnou hodnotu provést nějakou akci. Za příkaz Switch se do závorek uvede proměnná, která se má testovat. Do bloku Switch se pak uvádějí jednotlivé hodnoty a příslušené bloky kódu, které se provedou, pokud testovaná hodnota odpovídá uvedené hodnotě. Pokud testovaná proměnná neodpovídá žádné z uvedených hodnot a je použit nepovinný blok default, je vykonán kód v tomto bloku. Naopak pokud proměnná odpovídá více podmínkám, jsou vykonány všechny příslušné bloky.

```
# Skript switch.ps1
$test = 3
Switch ($test) {
      1 { '$test = 1' } # Blok se vykoná,
      # pokud je proměnná $test rovna číslu 1
      2 { '$test = 2' } # Blok se vykoná,
      # pokud je proměnná $test rovna číslu 2
      3 { '$test = 3' } # Blok se vykoná,
      # pokud je proměnná $test rovna číslu 3
      default { 'default' } # Blok se vykoná,
      # pokud proměnná neodpovídá žádné výše uvedené hodnotě }
```

Pro porovnání proměnné s hodnotami je defaultně použit operátor -eq. Lze však použít i vlastní podmínky. Ty stačí uvést do složených závorek před blok, který se má vykonat v případě jejich splnění. Lze tak například porovnávat proměnnou s rozsahem hodnot. Porovnávaná hodnota je v příkazech zastoupena proměnnou $_.

```
# Skript switch_ext1.ps1

$test = 3
Switch ($test) {
      {($_ -ge 1) -and ($_ -lt 3)} { echo '1 <= $test < 3' }
      {($_ -ge 3) -and ($_ -lt 5)} { echo '3 <= $test < 6' }
      {$_ -ge 6} { echo '6 <= $test'}
}
```

Tento skript vypíše do konzole řetězec "3 <= $test < 6".

Jako podmínky lze také používat řetězce, řetězce s wildcard znaky a dokonce regulární výrazy. Výsledky porovnání pro regulární výrazy se uchovávají v hashovací tabulce $matches. Pro názornou ukázku použijeme následující skript.

```
# Skript switch_ext2.ps1

$retezec = "ahoj"
# použití case sensitive switche
Switch -case ($retezec) {
      "ahoj" { "Pozdrav s malými písmeny" }
      "AHOJ" { "Pozdrav s velkými písmeny" }
}

# použití wildcard znaků
Switch -wildcard ($retezec) {
      "a*" { "Řetězec začíná písmenem a" }
      "*a*" { "Řetězec obsahuje písmeno a" }
}

# použití regulárních výrazů
Switch -regex ($retezec) {
      "^a" { "Řetězec začíná řetězcem a: $($matches[0])" }
}
```

Výsledek:

```
Pozdrav s malými písmeny
Řetězec začíná písmenem a
Řetězec obsahuje písmeno a
Řetězec začíná řetězcem a: a
```

Jak již bylo řečeno v úvodu k tomuto příkazu a také jak je vidět v předchozím příkladu, pokud pro řídící proměnnou platí více podmínek, jsou vykonány všechny tyto bloky.

```
# Skript switch_more_match.ps1

$test = 3;
Switch($test) {
      { $_ -lt 1 } { echo "$test < 1" }
      { $_ -lt 2 } { echo "$test < 2" }
      { $_ -lt 3 } { echo "$test < 3" }
      { $_ -lt 4 } { echo "$test < 4" }
      { $_ -lt 5 } { echo "$test < 5" }
}
```

Výsledek skriptu:

```
PS> .\switch_more_match.ps1
3 < 4
3 < 5
```

Pokud chceme, aby proběhl pouze blok pro první odpovídající podmínku, je třeba použít příkaz break, kterému je věnována vlastní podkapitola na konci kapitoly ohledně cyklů.

#### *1.9.4.1 Bash*

Analogickou jazykovou konstrukcí je v jazyce Bash příkaz case. Tento příkaz porovnává specifikovanou proměnnou s definovanými vzory. Těch může být klidně i více najednou – potom se oddělují znakem |, tak jako je tomu v následujícím příkladě. Pokud proměnná neodpovídá žádnému definovanému vzoru, lze použít symbol *, který slouží ke stejnému účelu jako hodnota default v příkazu Switch v PowerShellu.

```
#!/bin/bash
# Skript case.sh

case $USER in
Jirka | Uzivatel)
      echo "Vítej, uživateli $USER";;
root)
      echo "Vítej, administrátore!";;
*)
      echo "Neznámý uživatel!";;
esac
```

Po spuštění tohoto skriptu uživatelem Jirka se zobrazí v konzoli řetězec „Vítej, uživateli Jirka“, protože hodnota proměnné $USER odpovídá vzoru Jirka.

## 1.10 Cykly

Cykly slouží v programovacích jazycích k opakování bloku příkazů. Můžeme je rozdělit na cykly s pevně daným počtem opakování (cykly for a foreach), u nichž je předem dáno kolikrát proběhnou, a cykly ukončené podmínkou (cykly while, do, repeat), které jsou ukončeny až při splnění/nesplnění nějaké uživatelem definované podmínky.

### 1.10.1 Cyklus For

For cyklus je cyklus s pevným počtem opakování. Již před samotným vykonáním cyklu se tedy ví, kolikrát má proběhnout. Počet opakování se určuje řídící proměnnou. PowerShell využívá pro for cyklus stejnou syntaxi jako například jazyky C/C++ a Java. Definice for cyklu se tedy skládá ze tří částí, a to z inicializace řídící proměnné, definice koncové podmínky a nastavení kroku o jakou hodnotu se řídící proměnná změní po vykonání jednoho cyklu.

Pro výpis čísel od jedné včetně do pěti včetně lze použít tento skript. Jedná se o nejjednodušší použití for cyklu.

```
# Skript for.ps1

For($i=1; $i -le 5; $i++) {
      $i
}
```

Výsledek:

```
1
2
3
4
5
```

#### *1.10.1.1 Bash*

Pro stejný účel lze v Bashi také použít cyklus for. Má ale jiný způsob zápisu než jeho protějšek v PowerShellu. Dokonce se dá zapsat i více možnými způsoby. Pro definování počtu opakování cyklu používá spíše výčet hodnot nebo rozsah hodnot pro řídící proměnnou. Cyklus se provede pro každou z těchto hodnot a má tak spíš blíže k Foreach cyklu PowerShellu, který bude zmíněn v následující kapitole. Také je velice podobný for cyklu v jazyce Python.

```
#!/bin/bash
# Skript for_list.sh

# Řídící proměnná definovaná výčtem prvků
for i in 1 2 3 4 5
do
      echo $i
done
```

```
#!/bin/bash
# Skript for_range.sh

# Řídící proměnná definovaná rozsahem hodnot – Bash od verze 3.0
for i in {1..5}
do
      echo $i
done
```

Výsledkem obou výše uvedených cyklů bude tento výpis:

```
1
2
3
4
5
```

Od verze Bash 4.0 lze specifikovat i o jakou hodnotu se mění hodnota v rozsahu. K tomu se použije definice rozsahu ve formátu {první hodnota..poslední hodnota..změna hodnoty}.

```
#!/bin/bash
# Skript for_range_step.sh

for i in {2..10..2}
do
      echo $i
done
```

Výsledkem je tedy výpis:

```
2
4
6
8
10
```

Klasický for cyklus, známý například z jazyka C, lze zapsat i v Bashi, a to tímto způsobem:

```
#!/bin/bash
# Skript for_c.sh

for (( i=1; i <= 5; i++ ))
do
      echo $i
done
```

Výsledek bude znovu výpis hodnot od 1 včetně do 5 včetně.

### 1.10.2 Cyklus Foreach

Tento cyklus je také cyklem s pevným počtem opakování, jelikož vychází z běžného for cyklu. Často se pro něj v jiných jazycích používá příkaz for, avšak s poněkud odlišnými parametry než obyčejně. Tento cyklus totiž zpravidla prochází prvky v poli/kolekci prvek po prvku a pro každý z nich provede příkazy v těle cyklu. V PowerShellu existují dvě varianty cyklu Foreach. První je objekt ForEach-Object, který přijímá data z roury a druhý je příkaz Foreach.

#### *1.10.2.1 ForEach-Object*

Tento cmdlet, reprezentující Foreach cyklus, přijímá objekty z roury a pro každý z nich vykoná specifikovaný blok příkazů. Tyto objekty jsou zpracovány ihned, jak dorazí z roury do cyklu. Avšak oproti příkazu Foreach je tento cyklus pomalejší. Hodí se proto pro průběžné zpracování dat ještě dříve, než skončí předchozí příkaz. To je nejlépe vidět, pokud je vykonáván cyklus pro mnoho objektů, které jsou získávány delší dobu. Dobrým příkladem je tento příkaz, který rekurzivně prochází adresáře a získává soubory z disku C: a okamžitě pro ně vypisuje kompletní cestu:

```
PS> dir C:\ -recurse | ForEach-Object { $_.FullName }
```

Kompletní cesty pro tyto soubory jsou za běhu vypisovány na výstup.

#### *1.10.2.2 Foreach*

Oproti cyklu v cmdletu ForEach-Object se chová cyklus Foreach odlišně. Výstup je sice stejný, avšak postup při zpracování příkazů se liší. Při použití tohoto cyklu jsou nejprve získána všechna data (při použití příkazu musí příkaz skončit a navrátit data) a teprve poté se provádějí příkazy pro operace s nimi. Analogickým příkladem k předchozímu příkladu pro cyklus ForEach-Object je tento příkaz:

```
PS> Foreach($soubor in dir C:\ -recurse) { $soubor.FullName }
```

Po spuštění tohoto příkazu se dlouhou dobu nic neděje a poté jsou vypsány kompletní cesty ke všem souborům na disku C:.

Tento cyklus je výhodnější používat, pokud jsou data již připravena, jelikož je podstatně rychlejší než ForEach-Object. Čas potřebný pro vykonání obou cyklů lze změřit příkazem Measure-Command. Jelikož je však čas, po který tyto cykly běží, značně proměnlivý, je zapotřebí provést několik opakování měření a vypočítat průměrné hodnoty naměřených časů. To vše je provedeno skriptem foreach_foreachobject.ps1, který je v příloze (Příloha D).

```
PS> .\foreach_foreachobject.ps1
Počet opakování: 50 a průměrný čas pro For-EachObject: 61.953486
Počet opakování: 50 a průměrný čas pro Foreach: 3.520164
```

Z výsledků jasně vyplývá, že cyklus Foreach je v porovnání s cyklem ForEach-Object při práci s již připravenými daty mnohonásobně rychlejší.

V Bashi je cyklus for v podstatě i cyklem Foreach. Více je napsáno v kapitole o for cyklech.

### 1.10.3 Cyklus While

Dalším cyklem, který je běžně používaný i v jiných programovacích jazycích, je cyklus while. Jedná se o cyklus ukončený podmínkou. Tento cyklus se používá v případě, že není dopředu známo, kolik opakování je potřeba provést, ale je známa podmínka, kdy se má provádění cyklu ukončit. Před provedením cyklu se otestuje, zda definovaná podmínka platí. Pokud platí, cyklus proběhne, pokud neplatí, cyklus neproběhne a je ukončen.

#### 1.10.3.1 PowerShell

Použití cyklu while si ukážeme na stejném jednoduchém příkladu jako v případě for cyklu, aby vynikly rozdíly mezi jednotlivými cykly. Tento skript vypíše na obrazovku čísla od jedné včetně do pěti včetně.

```
# Skript while.ps1

$i = 1
While($i -le 5) { # Podmínka – proměnná $i je menší nebo rovná pěti
      $i    # Vypsání proměnné $i
      $i++  # Zvýšení hodnoty proměnné $i o 1
}
```

K tomuto účelu by však lépe posloužil for cyklus, jelikož počet opakování cyklu je předem známý. Cyklus while se používá spíše v jiných případech, například když se čtou data ze souboru. V tomto příkladě použijeme třídu System.IO.File z prostředí .NET, která obsahuje statické metody pro práci se soubory, a to zejména metodu OpenText pro otevření textového souboru v kódování UTF-8 [31]. Také je použita třída System.IO.StreamReader (v ukázce proměnná $soubor), která obsahuje metodu ReadLine pro čtení řádku z textového souboru [32].

```
# Skript while_read.ps1

# Zapsání zkušebního textu do souboru test.txt
"Ahoj, světe," | Out-File "$pwd\test.txt"
"jak se vede?" | Out-File -FilePath "$pwd\test.txt" -Append

# Otevření souboru a přiřazení do proměnné
$soubor = [system.io.file]::OpenText("$pwd\test.txt")
# Pokud soubor není na konci, provede se cyklus
While(!($soubor.EndOfStream)) {
      # Přečtení řádku souboru
      $soubor.ReadLine()
}
# Zavření souboru
$soubor.close()
```

Výsledkem spuštění tohoto skriptu je výpis obsahu souboru test.txt na obrazovku.

```
Ahoj, světe,
jak se vede?
```

#### *1.10.3.2 Bash*

V Bashi je cyklus while samozřejmě také. Avšak opět jako v případě podmínek if používá pro vyhodnocení řídící podmínky hranaté (případně dvojité hranaté či dvojité kulaté) závorky. Také lze místo podmínky použít příkaz. V tom případě se bude o proběhnutí cyklu rozhodovat podle návratové hodnoty tohoto příkazu. Pokud příkaz skončí úspěšně, cyklus proběhne.

Analogicky k předchozímu příkladu v PowerShellu, který vypisoval čísla od jedné včetně do pěti včetně, si uvedeme tento skript v Bashi se stejnou funkcí.

```
#!/bin/bash
# Skript while.sh

i=1
while (( $i <= 5 )); do
      echo $i
      ((i++))
done
```

Zvláštností oproti ostatním programovacím jazykům je v Bashi cyklus until. Jedná se v podstatě o cyklus while s opačnou podmínkou na začátku. Cyklus se tedy vykonává, dokud není tato podmínka splněna. Praktické využití tohoto cyklu si ukážeme ve skriptu.

```
#!/bin/bash
# Skript until.sh

q=""
until [[ "$q" == "q" ]]; do
      echo "Stiskněte znak q"
      read q
done
echo "q stisknuto, končím!"
```

Při spuštění tohoto skriptu je uživatel požádán o stisknutí klávesy q a je mu dáno tolik pokusů, dokud tuto klávesu nestiskne. Cyklus končí ve chvíli, kdy je klávesa q stisknuta. Spuštění tohoto skriptu vypadá následovně.

```
$ ./until.sh
Stiskněte znak q
n
Stiskněte znak q
o
Stiskněte znak q
q
q stisknuto, končím!
```

Pro tento účel by bylo spíše vhodné použít cyklus do while, který je představen v následující kapitole.

### 1.10.4 Cyklus Do

Cyklus do, neboli také do while cyklus, je obdobou cyklu while. Na rozdíl od něj je však ukončovací podmínka uvedena a testována až na konci cyklu, což znamená, že cyklus vždy alespoň jednou proběhne. Proto se hodí zejména ke zpracování uživatelského vstupu v případě, že je uživateli dána možnost zadávat hodnoty, dokud nesplní určitou podmínku.

To si ukážeme na následujícím skriptu.

```
# Skript do_while.ps1

do {
      $q = Read-Host 'Stiskněte znak q'
} while ($q -ne 'q')
echo "q stisknuto, končím!"
```

```
PS> .\do_while.ps1
Stiskněte znak q: t
Stiskněte znak q: j
Stiskněte znak q: o
Stiskněte znak q: q
q stisknuto, končím!
```

Tento cyklus existuje jak v Jazyce C, tak i v Jazyce Java a většině dalších jazyků. Obdobný cyklus repeat until je i v jazyce Pascal, avšak ten má opačnou ukončovací podmínku – cyklus se zastaví v případě splnění podmínky. Bash však žádný podobný cyklus nemá.

### 1.10.5 Switch jako cyklus

V jazyce PowerShell je speciálním cyklem i příkaz switch. Ten dokáže provádět rozhodování i pro pole prvků. Jednotlivé prvky jsou pak v samotném příkazu switch přístupné přes proměnnou $_. Lze tak například vypsat čísla od jedné včetně do pěti včetně tímto skriptem.

```
# Skript switch_loop.ps1

$a = 1..5
Switch($a) {
      Default { "$_" }
}
```

```
PS> .\switch_loop.ps1
1
2
3
4
5
```

### 1.10.6 Řízení cyklů

V případě, že je potřeba změnit běh cyklu během jeho vykonávání, jsou k dispozici dva příkazy pro řízení cyklů, známé i z jiných programovacích jazyků. Tyto příkazy lze použít v jakémkoliv cyklu na kterémkoliv místě. Prvním takovým příkazem je příkaz break.

#### *1.10.6.1 Break*

Příkaz break slouží k přerušení vykonávání cyklu a „vyskočení“ z cyklu ven. Kód, který následuje v cyklu po tomto příkazu, se tedy nevykoná. Pokračuje se vykonáváním kódu za cyklem. Při použití v příkazu switch se stejným způsobem „vyskočí“ z celého příkazu switch. Použití vypadá následovně:

```
# Skript break.ps1

$x = 0;
while($x -le 5) {
      $x++
      $x
      if($x -eq 3) {
            break
      }
}
echo $('$x je rovno {0}' -f $x)
```

Příkaz break se provede, pokud se proměnná $x rovná číslu 3. Bez tohoto příkazu by se cyklus zastavil, až pokud by se hodnota $x rovnala číslu 5. Při spuštění skriptu však vidíme, že hodnota $x je na konci skriptu rovna číslu 3.

```
PS> .\break.ps1
1
2
3
$x je rovno 3
```

Při použití vnořených cyklů lze také specifikovat, pro který cyklus má příkaz break platit. Jinak je totiž ukončen nejbližší cyklus, ze kterého je tento příkaz zavolán. Pokud chceme ukončit některý ze „vzdálenějších“ cyklů, je potřeba tento cyklus označit štítkem. Tento štítek mohou mít pouze cykly. V následujícím příkladu jsou ukončeny oba cykly, pokud jsou proměnné $x a $i rovny jedné. [33]

```
# Skript break_nested_loop.ps1

:outer_for For($i = 0; $i -lt 5; $i++) {
      $x = 0
      While($x -lt 2) {
            echo $('$x je rovno {0} a $i je rovno {1}' -f $x, $i)
            if(($i -eq 1) -and ($x -eq 1)) {
                  echo 'Konec všech cyklů'
                  break outer_for
            }
            $x++
      }
}
```

Výsledek:

```
PS> .\break_nested_loop.ps1
$x je rovno 0 a $i je rovno 0
$x je rovno 1 a $i je rovno 0
$x je rovno 0 a $i je rovno 1
$x je rovno 1 a $i je rovno 1
Konec všech cyklů
```

Další využití tohoto příkazu je, jak již bylo zmíněno, v blocích obsažených v příkazu switch. Pokud je totiž více podmínek platných, provedou se bloky pro všechny podmínky. V tomto příkladu použijeme upravený skript (switch_more_match.ps1) z kapitoly o příkazu switch a zabráníme vykonání více bloků.

```
# Skript break_switch_more_match.ps1

$test = 3;
Switch($test) {
      { $_ -lt 1 } { echo "$test < 1"; break }
      { $_ -lt 2 } { echo "$test < 2"; break }
      { $_ -lt 3 } { echo "$test < 3"; break }
      { $_ -lt 4 } { echo "$test < 4"; break }
      { $_ -lt 5 } { echo "$test < 5" }
}
```

Skript spustíme a ověříme, že se na výstupu zobrazí pouze jeden řádek.

```
PS> .\break_switch_more_match.ps1
3 < 4
```

To je způsobeno tím, že po vykonání prvního odpovídajícího bloku dojde k opuštění celého bloku switch díky příkazu break.

#### 1.10.6.1.1 Bash

V jazyku Bash je příkaz break používán stejným způsobem a se stejným významem. Jediné dva rozdíly jsou ty, že příkaz case je ukončen dvěma středníky namísto příkazu break a také pokud je příkaz break použit ve vnořených cyklech, a je třeba opustit některý ze „vzdálenějších cyklů“, nepoužívají se jmenovky pro označení opouštěného cyklu, ale za příkaz break se uvede číslo, označující kolikátý cyklus se má opustit. Toto číslo musí být větší nebo rovno číslu jedna. Ukázkový skript ukončí oba cykly, pokud je proměnná $y rovna číslu tři.

```
#!/bin/bash
# Skript break.sh

for x in {0..5}; do
      for y in {5..0}; do
            echo "x = $x, y = $y"
            if (( $y == 3 )); then
                  echo "Konec všech cyklů"
                  break 2
            fi
      done
done
```

Výsledek skriptu:

```
$ ./break.sh
x = 0, y = 5
x = 0, y = 4
x = 0, y = 3
Konec všech cyklů
```

### *1.10.6.2 Continue*

Kromě opuštění cyklu příkazem break lze také vynutit přeskočení zbytku bloku současného opakování cyklu a provedení následujícího opakování cyklu příkazem continue. To se špatně popisuje slovy, a proto je lepší uvést si příklad.

```
# Skript continue.ps1

$x = 0;
while($x -le 5) {
      $x++
      if($x -eq 3) {
            continue
      }
      $x
}
```

Tento cyklus vypíše na obrazovku následující výstup:

```
PS> .\continue.ps1
1
2
4
5
6
```

Je vidět, že číslo 3 se na obrazovku nevypíše. To je dáno tím, že příkaz continue se provede, pokud je proměnná $x rovna číslu tři. A jelikož je tento příkaz ještě před výpisem proměnné $x na obrazovku, příkaz pro výpis se přeskočí a cyklus začne další opakování.

V Bashi je příkaz continue totožný se svým protějškem v jazyku PowerShell jak z hlediska významu, tak i z hlediska syntaxe. A stejně tak, jako u příkazu break, pokud se za něj uvede číslo, bude označovat, pro kolikátý cyklus platí.

## 1.11 Skripty

Interaktivní shell je velice užitečný pro správu počítače. Zejména pokud je potřeba něco okamžitě zjistit nebo změnit, stačí zadat příslušné příkazy a ty jsou vykonány. Někdy však nestačí pouhé zadávání jednotlivých a poměrně jednoduchých příkazů do konzole, ale je potřeba vykonat podstatně složitější operaci nebo tuto operaci vykonávat častěji. Nebo dokonce tuto operaci provádět na několika různých počítačích. Proto by se hodilo, kdyby šlo tyto příkazy nějakým způsobem uchovat a v případě potřeby je rychle použít. K tomu slouží skripty. To jsou obyčejné textové soubory (často mívají nějakou typickou příponu pro daný skriptovací jazyk), které uchovávají posloupnosti příkazů – v podstatě se jedná o takové funkce uložené na disku na rozdíl od funkcí uložených v paměti. Skripty jsou pak čteny interpretem a příkazy v nich postupně vykonávány stejně tak, jako by byly psány postupně do příkazové řádky. Pokud jsou ve skriptu použity funkce, měly by být ve skriptu uvedeny co nejvýše – funkce lze používat až potom, co jsou ve skriptu definovány.

### 1.11.1 PowerShell

V PowerShellu jsou skripty textové soubory s koncovkou .ps1. Práce s nimi je prakticky stejná jako práce s funkcemi. Lze například spouštět skripty bez parametrů, které stále provádějí jednu a tu samou posloupnost příkazů, nebo lze vytvářet skripty, které mají vstupní parametry. Tyto parametry se deklarují stejným způsobem jako u funkcí.

#### *1.11.1.1 Zabezpečení skriptů*

Tato kapitola čerpá ze zdrojů [34] [35]. Jednoduchý skript, který vytiskne na výstup datum, může vypadat takto:

```
# Skript datum.ps1
Get-Date
```

Při pokusu spustit první skript však může uživatele čekat nemilé překvapení v podobě chybové hlášky:

```
PS> .\datum.ps1
…\datum.ps1 : File …\datum.ps1 cannot be loaded because running scripts
is disabled on this system. For more information, see
about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ …\datum.ps1
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

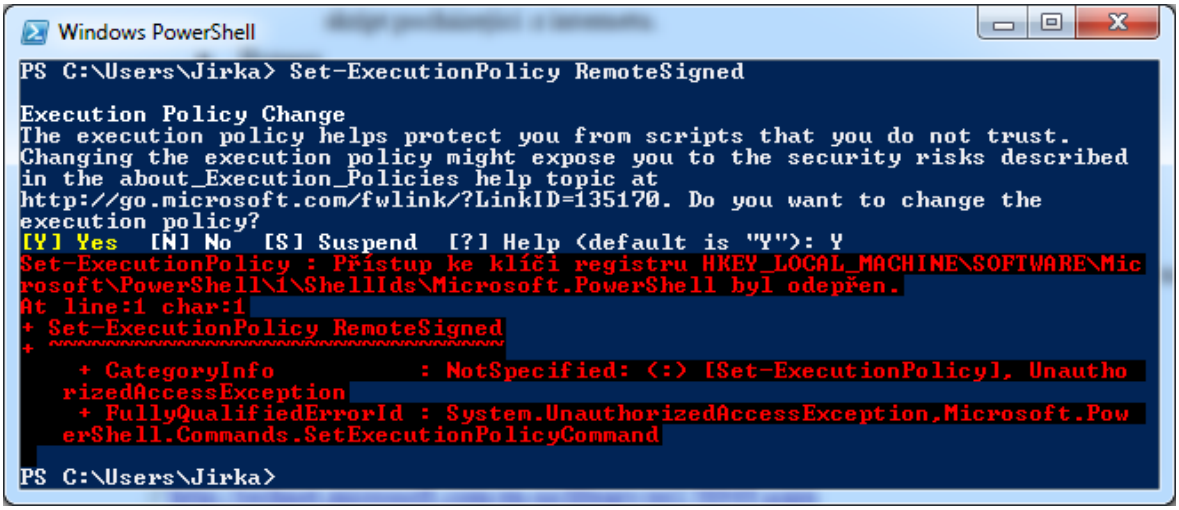
Tento chybový výpis znamená, že v PowerShellu není povoleno spouštění skriptů. Z důvodu bezpečnosti je totiž ve výchozím nastavení zakázáno. To si můžeme ověřit příkazem Get-ExecutionPolicy.

```
PS> Get-ExecutionPolicy
Restricted
```

Z výsledku tohoto příkazu je vidět, že politika spouštění skriptů je nastavena na hodnotu Restricted, což v překladu do češtiny znamená slovo „omezený“. Politika spouštění skriptů může nabývat více hodnot, než jen povolení skriptů a zakázání skriptů. Tyto hodnoty jsou:

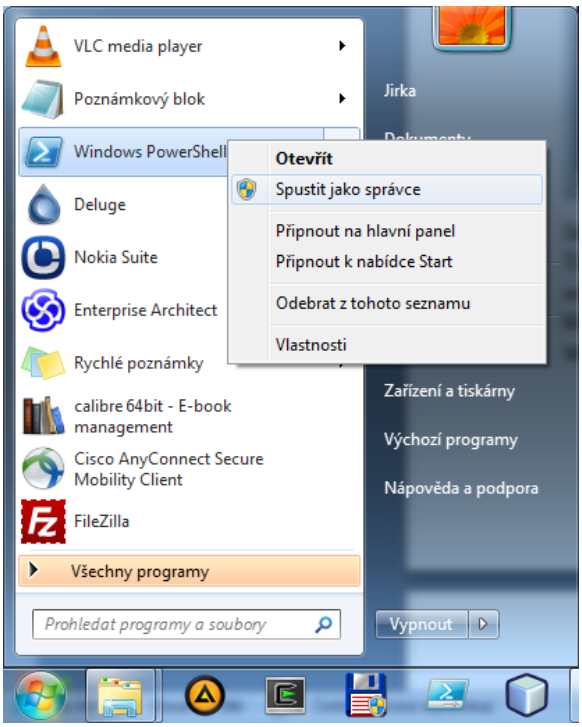
- Restricted
  - Povoleny pouze příkazy v interaktivním shellu. Vykonávání skriptů je zakázáno.
- AllSigned
  - Povolí spuštění pouze pro digitálně podepsané skripty.
- RemoteSigned
  - Povolí spouštění pro lokální skripty (nepocházející z internetu). Skripty pocházející z internetu musí být digitálně podepsané.
- Unrestricted
  - Povolí spuštění všech skriptů, ale varuje uživatele, pokud chce spustit skript pocházející z internetu.
- Bypass
  - Spouštění všech skriptů je povoleno.
- Undefined
  - Není nastavena žádná politika spouštění skriptů. Použije se politika Restricted.

Pro spuštění výše uvedeného skriptu je potřeba změnit politika spouštění skriptů nejlépe na hodnotu RemoteSigned. To provedeme příkazem Set-ExecutionPolicy.



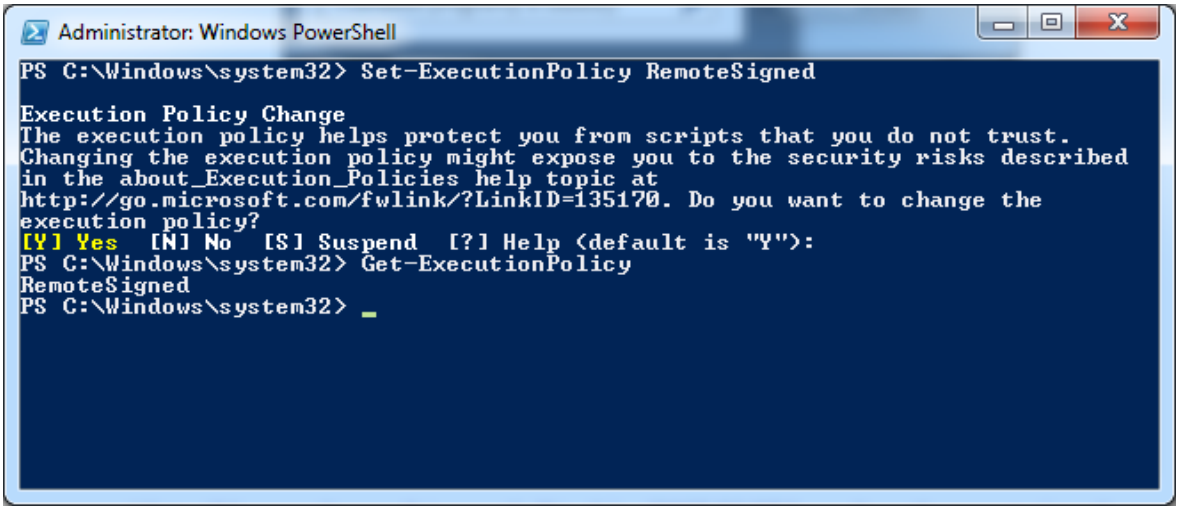

**Obrázek 1 – Nedostatečné oprávnění pro změnu politiky spouštění skriptu**

Příkaz se zeptá, zda chceme změnit politiku spuštění skriptů. Stačí stisknout klávesu enter, jelikož výchozí odpověď je souhlas. Avšak jak je vidět na obrázku (Obrázek 1), pokus změnit politiku spouštění skriptů skončí chybou. To je způsobeno tím, že příkaz Set-ExecutionPolicy potřebuje ke své funkci práva administrátora. Je proto třeba spustit PowerShell se zvýšeným oprávněním. To provedeme kliknutím pravým tlačítkem na ikonu PowerShellu a zvolením tlačítka Spustit jako správce. Viz Obrázek 2.




**Obrázek 2 – Spuštění PowerShellu s právy administrátora**

Potvrdíme okno, které se objeví, a v konzoli s administrátorskými právy znovu spustíme příkaz Set-ExecutionPolicy s argumentem RemoteSigned. Tentokrát příkaz proběhne bez problémů a příkazem Get-ExecutionPolicy ověříme, že se hodnota politiky spouštění skriptů změnila na námi požadovanou hodnotu. Toto vše je vidět na následujícím obrázku (Obrázek 3).




**Obrázek 3 – Úspěšná změna politiky spouštění skriptů**

Nyní je možné bez problémů spustit náš skript pro výpis data.

```
PS> .\datum.ps1

20. dubna 2013 14:37:28
```

### *1.11.1.2 Parametry skriptů*

Jak již bylo zmíněno, skriptům lze předávat parametry jako funkcím. Všechny parametry jsou buďto uloženy v poli $args, nebo se na začátku skriptu definují pojmenované parametry klíčovým slovem Param. Takový skript potom vypadá takto:

```
# Skript params.ps1

Param([String]$jmeno, [String]$prijmeni)
echo "Jméno: $jmeno, příjmení: $prijmeni a parametry navíc jsou: '$args'"
```

Spuštěním tohoto skriptu a předáním argumentů získáme následující výstup:

```
PS> .\params.ps1 Jiří Kratochvíl nadbytečné parametry 1 2 3
Jméno: Jiří, příjmení: Kratochvíl a parametry navíc jsou: 'nadbytečné
parametry 1 2 3'
```

Více informací k parametrům je uvedeno v kapitole o funkcích.

#### *1.11.1.3 Pokročilejší skripty*

Jako ukázku složitějšího skriptu si můžeme ukázat například skript, který je v programu PSConfig používán k nastavování síťových adaptérů. Ten je okomentovaný obsažen v příloze (Příloha E).

### 1.11.2 Bash

V Bashi jsou skripty textové soubory, které ani nemusí mít příponu. Často se však uvádí přípona .sh pro jednodušší identifikaci. Práce se skripty je také podobná práci s běžnými příkazy. Parametry se uvádí za volání skriptu.

#### *1.11.2.1 Zabezpečení skriptu*

Oproti PowerShellu nelze zakázat spouštění skriptu, pokud ho lze přečíst. Také neexistuje žádná kontrola digitálních podpisů pro spouštění skriptů. S právy pro spouštění skriptů se to má následujícím způsobem:

**Tabulka 3 – Nastavení práv souboru skriptu v Bashi**

| Právo read | Právo execute | Výsledek |
|---|---|---|
| **0** | **0** | Skript nelze spustit. |
| **1** | **0** | Skript lze spustit příkazem "bash *název skriptu*". |
| **0** | **1** | Skript nelze spustit. |
| **1** | **1** | Skript lze spustit příkazem "bash *název skriptu*" i "*./název skriptu*" |

Pro spuštění skriptu je tedy hlavně potřeba, aby měl nastavené příslušné právo pro čtení.

#### *1.11.2.2 Parametry skriptů*

Skripty v Bashi mají k funkcím stejný vztah tak, jako mají skripty k funkcím v PowerShellu. Proto se naprosto stejným způsobem předávají parametry jak u skriptů, tak u funkcí. Stačí ukázka krátkého skriptu využívajícího parametry. Podrobnější popis je v kapitole o funkcích.

```
#!/bin/bash
# Skript script.sh

echo "Skript spuštěn jako: $0"
for paramat in "$@"; do
      echo "\$@: $paramat"
done
```

Spuštění a výstup vypsaný na obrazovku vypadá takto:

```
$ ./script.sh raz dva "Ahoj světe"
Skript spuštěn jako: ./script.sh
$@: raz
$@: dva
$@: Ahoj světe
```

# 2 Windows Management Instrumentation (WMI)

„Windows Management Instrumentation je infrastruktura pro správu dat a operací na systémech typu Windows.“ [36]

Jedná se o implementaci Web-Based Enterprise Management (WBEM) [8], což je norma mající za úkol sjednotit způsob přístupu k systémovým informacím ve firemním prostředí. Součástí WBEM je také Common Information Model (CIM). „CIM je rozšiřitelný, objektově orientovaný datový model, který obsahuje informace o různých částech společnosti. [37]“ To znamená, že lze vytvářet různé třídy, které mohou reprezentovat například pevné disky, aplikace nebo jakákoliv jiná zařízení. Z těchto objektů lze poté získávat informace o daných zařízeních nebo operacemi s CIM objekty příslušná zařízení i nastavovat. CIM není vázán na žádný operační systém.

CIM definuje tři úrovně tříd [37]:

- Core
    - Core třídy reprezentují objekty, které jsou potřebné pro správu obecně. Jsou to abstraktní třídy. Příkladem Core třídy je třída __Parameters.
- Common
    - Common třídy reprezentují objekty, které jsou specifické pro různé oblasti správy systémů. Tyto třídy jsou nezávislé na operačním systému a jsou abstraktními třídami. Jsou rozšířením Core tříd. Příkladem je například třída CIM_NetworkAdapter.
- Extended
    - Tyto třídy jsou rozšířené Common třídy. Jsou již závislé na použité platformě a tak existují zvlášť například pro Win32 prostředí nebo pro UNIX. Příkladem je třída Win32_NetworkAdapter.

WMI bylo poprvé představeno ve Windows 2000 a od té doby je součástí každé verze systému Windows. Jelikož WMI používá CIM, lze WMI použít jak pro získávání informací o počítači, tak k různému konfigurování počítače. Komponenty počítače a jeho vlastnosti jsou reprezentovány v různých třídách v souladu s WBEM. Třídy, které lze přímo používat pro správu počítače, mají prefix Win32, jelikož se jedná o Extended třídy. Obsahují tedy prakticky vše potřebné k obsluze daného zařízení na platformě Win32.

Ukážeme si, jakou má taková třída stavbu. Jelikož je později v praktické části práce používána třída Win32_NetworkAdapter, bude nejlepší použít jako ukázku právě ji. Tento kód je z oficiální dokumentace WMI[1]. Syntaxe ukázky je Managed Object Format, což je jazyk pro popis CIM tříd.

```
class Win32_NetworkAdapter : CIM_NetworkAdapter
{
  string    AdapterType;
  uint16    AdapterTypeID;
  boolean   AutoSense;
  uint16    Availability;
  string    Caption;
  uint32    ConfigManagerErrorCode;
  boolean   ConfigManagerUserConfig;
  string    CreationClassName;
  string    Description;
  string    DeviceID;
  boolean   ErrorCleared;
  string    ErrorDescription;
  string    GUID;
  uint32    Index;
  datetime  InstallDate;
  boolean   Installed;
  uint32    InterfaceIndex;
  uint32    LastErrorCode;
  string    MACAddress;
  string    Manufacturer;
  uint32    MaxNumberControlled;
  uint64    MaxSpeed;
  string    Name;
  string    NetConnectionID;
  uint16    NetConnectionStatus;
  boolean   NetEnabled;
  string    NetworkAddresses[];
  string    PermanentAddress;
  boolean   PhysicalAdapter;
  string    PNPDeviceID;
  uint16    PowerManagementCapabilities[];
  boolean   PowerManagementSupported;
  string    ProductName;
…
```

Hned z prvního řádku této ukázky je vidět, že Extended třída Win32_NetworkAdapter je odvozena od Common třídy CIM_NetworkAdapter. Syntaxe je totiž podobná jazyku C++ a třída uvedená za dvojtečkou je rodičovská třída [38]. Je vidět, že třída Win32_NetworkAdapter obsahuje mnoho atributů, které reprezentují konfiguraci síťové karty. S těmito atributy se pracuje skrze metody, které tato třída obsahuje pouze dvě, a to Reset pro reset logického zařízení a SetPowerState pro nastavení stavu napájení logického zařízení.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/aa394216%28v=vs.85%29.aspx [41]

V PowerShellu se příslušná instance WMI třídy získá přes cmdlet Get-WmiObject. Pro zobrazení všech WMI tříd v systému lze použít příkaz:

```
PS> Get-WmiObject -List
```

Pro získání instance třídy stačí zadat tento příkaz s názvem třídy požadované instance jako parametr -Class.

```
PS> Get-WmiObject -Class Win32_ComputerSystem
```

Tento příkaz vypíše na obrazovku informace o počítači.

```
Domain                  : WORKGROUP
Manufacturer            : SAMSUNG ELECTRONICS CO., LTD.
Model                   : R580
Name                    : TALIESIN
PrimaryOwnerName        : Jirka
TotalPhysicalMemory     : 4148744192
```

Toto však nejsou všechny atributy, které třída obsahuje. Všechny atributy lze zobrazit tímto příkazem:

```
PS> Get-WmiObject -Class Win32_ComputerSystem | Format-List *
```

Výpis je ale příliš dlouhý na to, aby byl zde uveden.

Při použití parametru -Class u cmdletu Get-WmiObject je také možné, že je získáno více instancí dané třídy. To se stane například v případě, že budeme chtít zjistit informace o síťových adaptérech tímto příkazem:

```
PS> Get-WmiObject -Class Win32_NetworkAdapter
```

Výše uvedený příkaz vrací objekty reprezentující všechny síťové adaptéry v počítači. To je ale nepraktické, pokud chceme vybrat pouze některé z nich podle námi požadované podmínky. Proto lze použít parametr -Query. Ten jako argument očekává řetězec v jazyku WMI Query Language (WQL) [39]. Tento jazyk vychází z SQL a slouží k získávání instancí tříd z WMI podle zadaných podmínek. Nejlepší bude názorná ukázka.

```
PS> Get-WmiObject -Query "SELECT * FROM Win32_NetworkAdapter WHERE
AdapterType='Ethernet 802.3' AND NetEnabled=true"
```

Tímto příkazem jsme získali všechny síťové adaptéry, jejichž atribut AdapterType je nastaven na hodnotu „Ethernet 802.3“ a NetEnabled na hodnotu true. To znamená, že jsme získali všechny aktivní Ethernetové adaptéry.

# 3 Program PSConfig

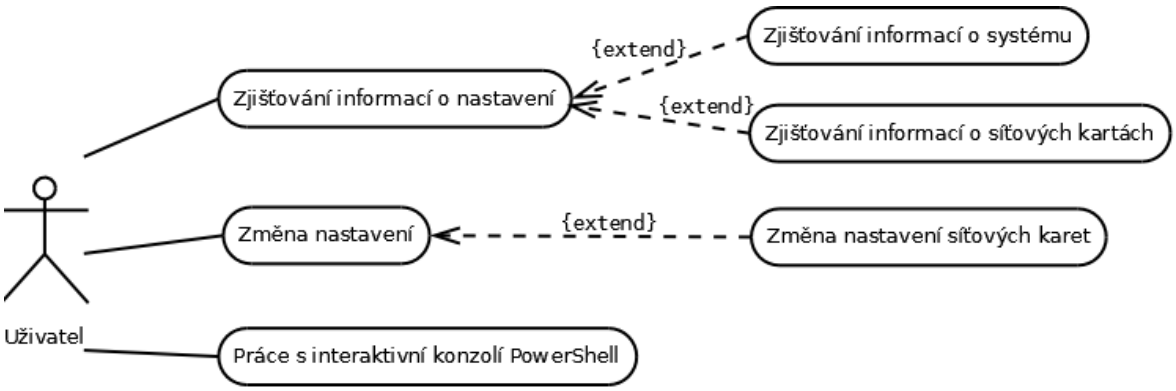
## 3.1 Popis aplikace

PSConfig je program napsaný v jazyce Java, který slouží k zobrazování informací o systému Microsoft Windows a také ke změnám systémových nastavení. K tomu je využíván program Windows PowerShell, s jehož procesem aplikace komunikuje.

## 3.2 Použití aplikace

Jak již bylo zmíněno, uživatel může program používat k získání informací o operačním systému a síťových adaptérech a také měnit jistá nastavení síťových adaptérů. U těch lze aktivovat DHCP nebo nastavit statickou IP adresu. K dispozici je také konzole PowerShellu, kde lze zadávat příkazy jako při normální práci s PowerShellem.

Úlohy, ke kterým lze program použít, jsou zobrazeny v tomto use case diagramu:



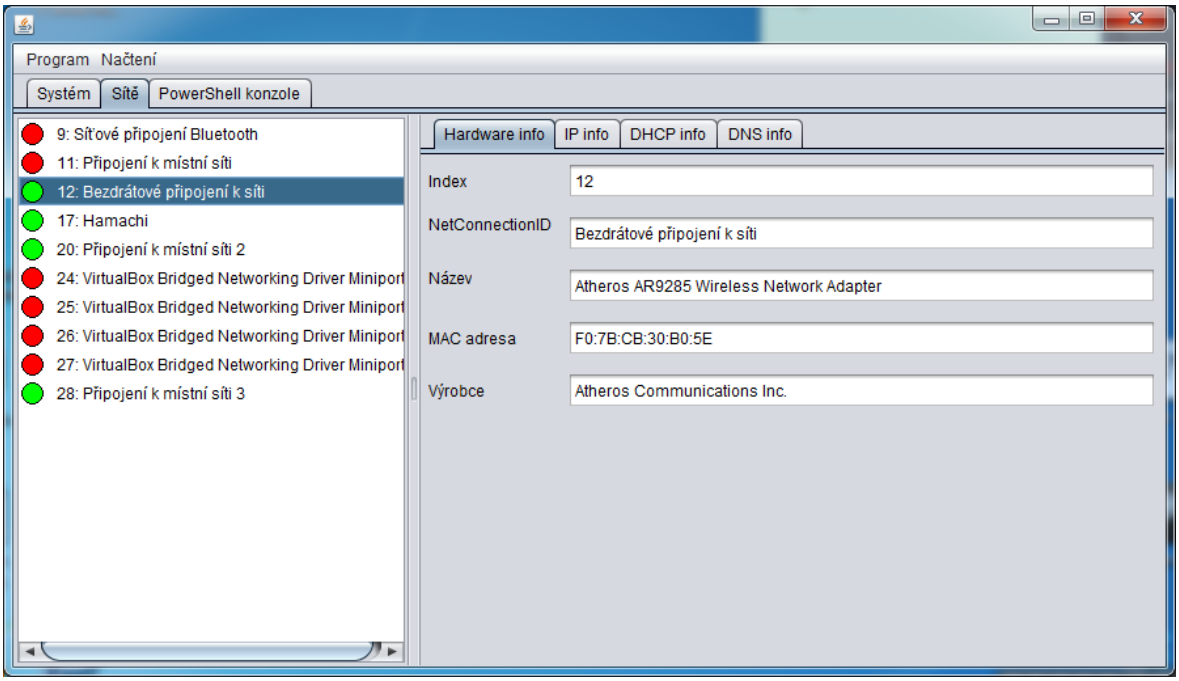

**Obrázek 4 – Use case diagram aplikace PSConfig**

### 3.2.1 Grafické rozhraní

Grafické rozhraní programu je vytvořeno tak, aby bylo co nejjednodušší a co nejpřehlednější. Proto je využito především javovské komponenty JTabbedPane, díky které jsou grafické komponenty rozmístěny na jednotlivých kartách a uživatel tak má stále přehled, ve které části programu se nachází. Hlavní karty existují tři.

Hlavní karty:

- Systém
  - Zobrazuje informace o systému.
- Sítě
  - Informace o nastavení síťových adaptérů. Některá nastavení lze měnit.
- PowerShell konzole
  - Interaktivní konzole PowerShellu.




**Obrázek 5 – Ukázka grafického rozhraní aplikace PSConfig**

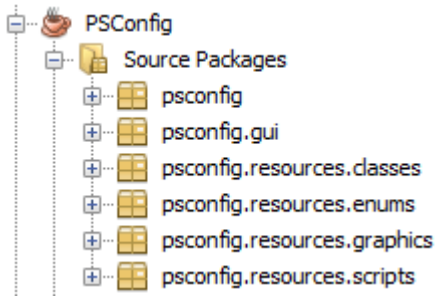
## 3.3 Struktura projektu

Program PSConfig byl vyvinut v prostředí NetBeans 7.3. Proto je projekt rozdělen do balíčků, kterým odpovídá i adresářová struktura celého projektu.

Projekt obsahuje následující balíčky/složky:

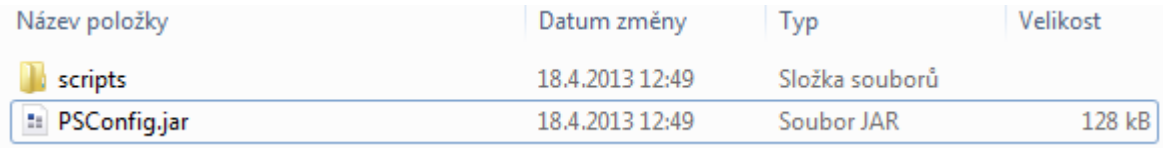
- psconfig
  - Kořenová složka obsahující zdrojový soubor pro hlavní okno aplikace a zbylé adresáře.
- psconfig/gui
  - Zdrojové soubory pro grafické rozhraní.
- psconfig/resources
  - Složka, ve které jsou další složky s více specializovanými soubory.
- psconfig/resources/classes
  - Negrafické třídy využité v projektu.
- psconfig/resources/enums
  - Výčtové typy.
- psconfig/resources/graphics
  - Soubory pro grafické rozhraní.
- psconfig/resources/scripts
  - Skripty pro PowerShell.

Uspořádání balíčků v programu NetBeans je vidět na tomto obrázku (Obrázek 6):




**Obrázek 6 – Struktura balíčků v aplikaci Netbeans**

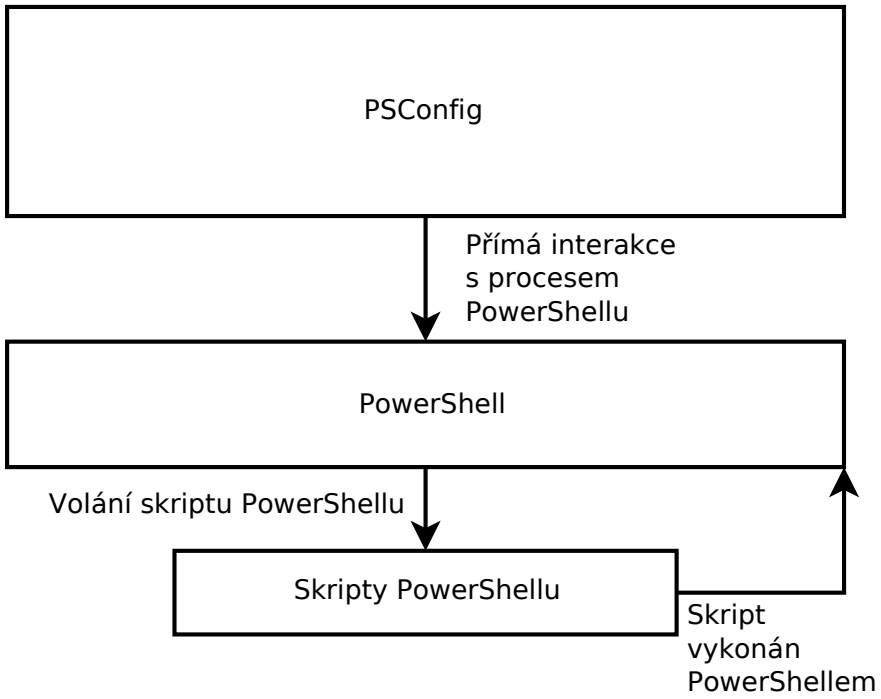
Ve výsledné aplikaci musí být skripty ve složce scripts, umístěné ve stejné složce jako samotný spustitelný soubor PSConfig.jar. Viz Obrázek 7.




**Obrázek 7 – Umístění skriptů v adresáři s aplikací PSConfig**

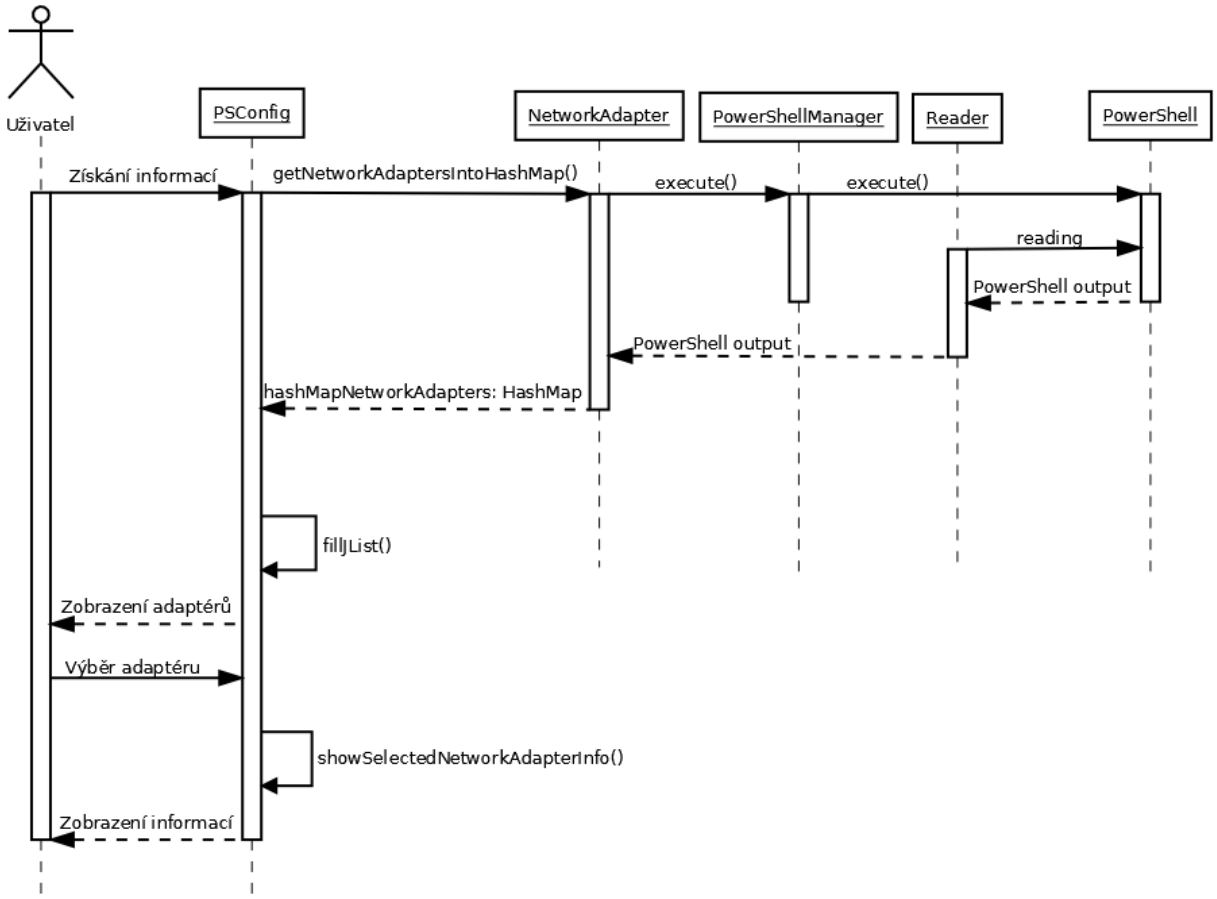
## 3.4 Architektura aplikace

PSConfig je samostatná aplikace využívající programu Windows PowerShell. Jak je vidět na obrázku (Obrázek 8), PSConfig vždy komunikuje přímo s procesem PowerShellu a buďto předává jednotlivé příkazy k vykonání nebo spustí skript, který PowerShell vykoná. Vstup do PowerShellu je přes standardní vstup jeho procesu a výstup je čten ze standardního výstupu procesu.




**Obrázek 8 – Architektura aplikace**

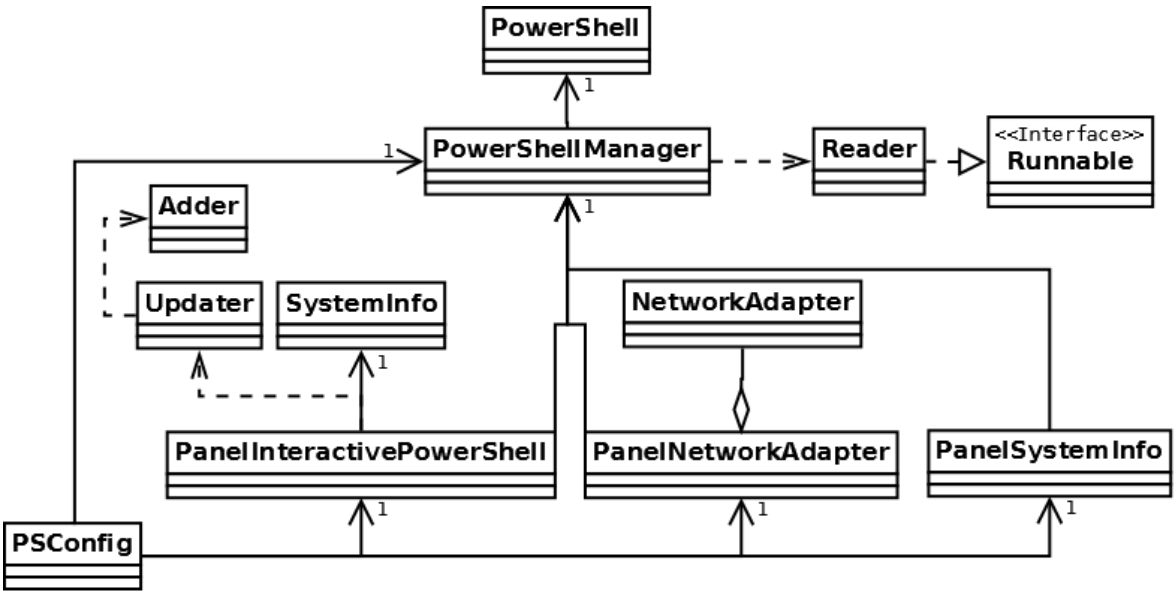
Jak přibližně probíhají jednotlivé úkony uvnitř programu, když chce uživatel zobrazit informace o jednotlivých síťových adaptérech, je vidět na obrázku (Obrázek 9). Zde je vyobrazeno, jak spolu komunikují objekty jednotlivých tříd.




**Obrázek 9 – Sequence diagram čtení nastavení síťových adaptérů**

### 3.4.1 Třídy

Závislosti tříd v programu PSConfig jsou znázorněny na UML diagramu na tomto obrázku (Obrázek 10):




**Obrázek 10 – UML diagram závislostí tříd v aplikaci**

#### *3.4.1.1 GUI třídy*

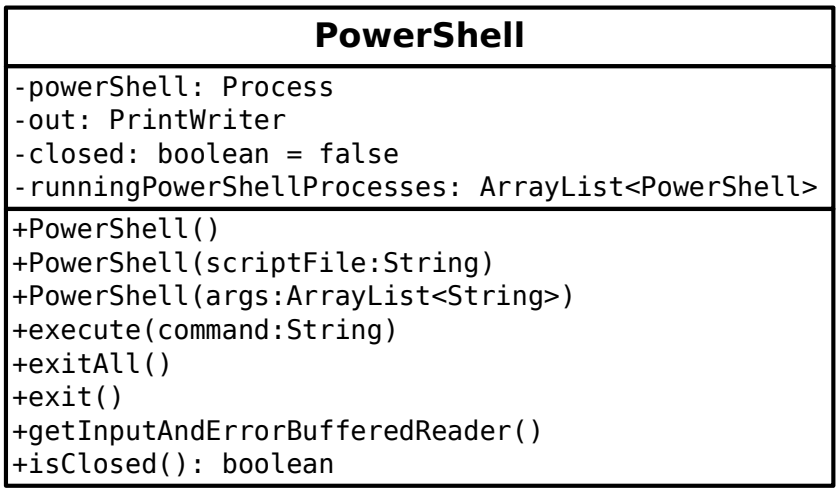
Hlavní část grafického rozhraní sestává z panelů umístěných na komponentě JTabbedPane. Každý z těchto panelů je instancí samostatné třídy odvozené od třídy JPanel a uzpůsobené k jejímu účelu. Tyto třídy jsou čtyři.

- PanelInteractivePowerShell
  - Zobrazuje komponenty pro přímou interakci s PowerShellem.
- PanelNetworkAdapter
  - Komponenty s informacemi o síťových kartách.
- PanelSystemInfo
  - Komponenty s informacemi o systému.
- PanelLoading
  - Panel, který se zobrazuje během načítání informací.

#### *3.4.1.2 Hlavní třídy aplikace*

V této podkapitole jsou popsány hlavní třídy aplikace. U každé z nich jsou popsány pouze nejdůležitější atributy a metody. Podrobnější popis je k dispozici v komentářích ve zdrojovém kódu.

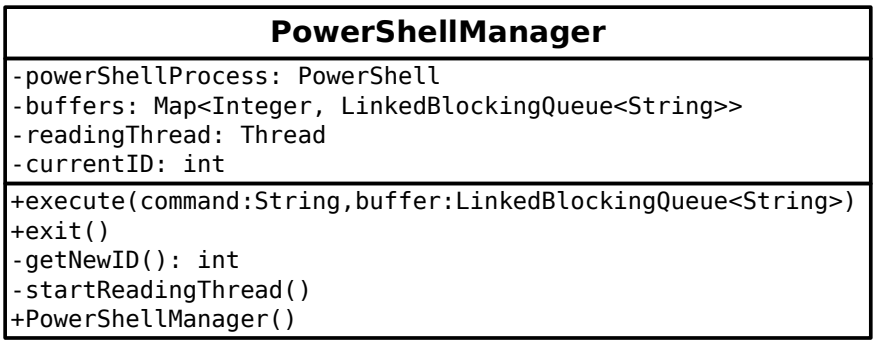
##### 3.4.1.2.1 PowerShell




**Obrázek 11 – UML diagram třídy PowerShell**

Tato třída reprezentuje proces PowerShellu v aplikaci PSConfig a slouží jako obalová třída pro instanci procesu PowerShellu.

#### 3.4.1.2.2 PowerShellManager

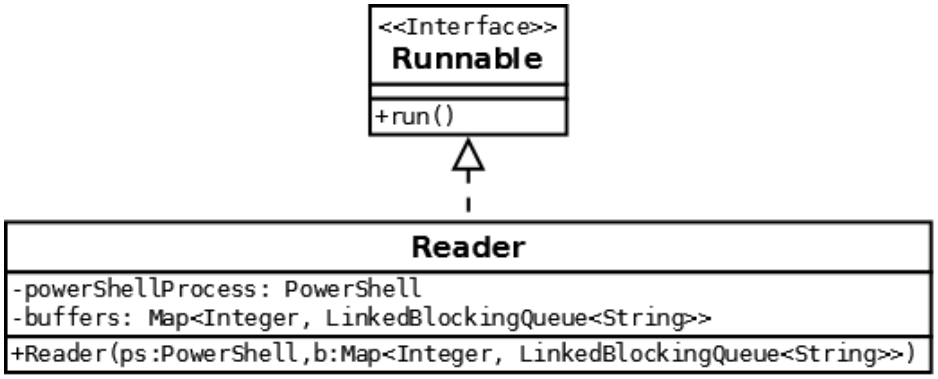



**Obrázek 12 – UML diagram třídy PowerShellManager**

Třída obstarávající prostředky pro práci se třídou PowerShell. Veškerý výstup z procesu, který je reakcí na příkaz poslaný PowerShellu, je ukládán do příslušného bufferu – instance třídy LinkedBlockingQueue. Třída PowerShellManager obstarává multiplexování vstupu do a výstupu z procesu PowerShellu. Je tak možné mít pouze jeden proces PowerShellu, používat ho z několika míst a přesto získávat výstup do samostatného bufferu, jako by procesů PowerShellu běželo víc.

Provedení příkazu PowerShellu a získání výsledku je s touto třídou jednoduché. Stačí zavolat metodu execute(command:String, buffer:LinkedBlockingQueue<String>). Jako parametry jí předáme vykonávaný příkaz (parametr command) a buffer, do kterého se má uložit výstup z PowerShellu (parametr buffer).

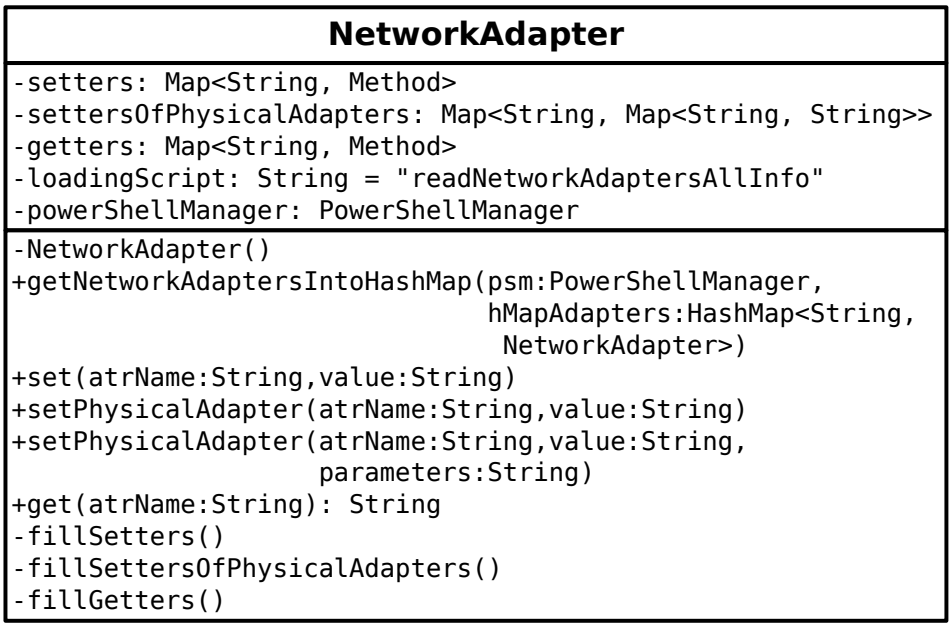
#### 3.4.1.2.3 Reader




**Obrázek 13 – UML diagram třídy Reader**

Třída Reader slouží k ukládání výstupu z procesu PowerShellu do příslušných bufferů. Implementuje rozhraní Runnable, jedná se tedy o třídu, jejíž instance běží jako vlákno.

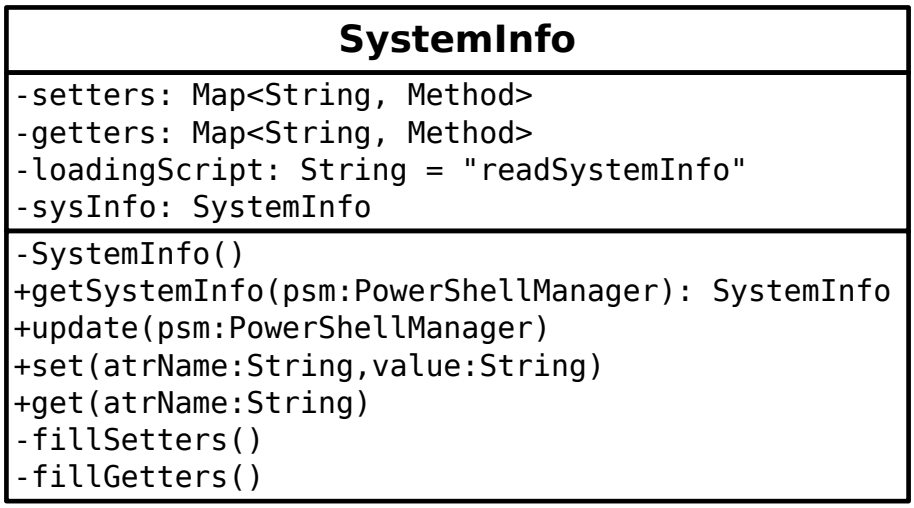
#### 3.4.1.2.4 NetworkAdapter




**Obrázek 14 – UML Diagram třídy NetworkAdapter**

Třída NetworkAdapter slouží k uchovávání informací o jednotlivých síťových adaptérech a také k manipulaci s těmito informacemi a se skutečnými síťovými adaptéry. Obsahuje mnoho atributů, reprezentujících vybrané atributy WMI třídy Win32_NetworkAdapter. Tyto atributy jsou z UML diagramu vynechány, jelikož je jich velké množství. Stejně tak jsou vynechány metody, které se k těmto atributům vážou, jako jsou settery a gettery.

#### 3.4.1.2.5 SystemInfo




**Obrázek 15 – UML diagram třídy SystemInfo**

SystemInfo je třída, která v sobě uchovává informace ohledně operačního systému. Její atributy představují vybrané atributy z WMI třídy Win32_ComputerSystem.

### 3.4.2 Skripty

Program PSConfig používá čtyři skripty PowerShellu.

- readNetworkAdaptersAllInfo.ps1
  - Skript sloužící k nahrání potřebných informací o síťových adaptérech do programu.
- readSystemInfo.ps1
  - Skript zobrazuje informace o počítači a systému. Ty jsou nahrány do programu, kde jsou zobrazeny.
- runScriptElevated.ps1
  - Skript pro spouštění jiných skriptů s právy administrátora.
- setNetAdapter.ps1
  - Skript, který mění nastavení síťových adaptérů.

#### 3.4.2.1 Skript readNetworkAdaptersAllInfo.ps1

Tento skript slouží k získání podrobných informací o síťových adaptérech, které jsou pak zobrazovány v programu PSConfig na kartě Sítě. Tyto informace jsou získávány z objektů WMI tříd Win32_NetworkAdapter a Win32_NetworkAdapterConfiguration. Srdcem celého skriptu je tato část:

```
…
Get-WmiObject -query `
'select * from Win32_NetworkAdapter where AdapterType="Ethernet 802.3"' |
ForEach-Object {

    $_
    $(Get-WmiObject -query `
      $('select * from Win32_NetworkAdapterConfiguration where
Index="{0}"' -f $_.Index))

} | Format-List Index, Name, NetEnabled, NetConnectionID, MACAddress,
Manufacturer, IPAddress, IPSubnet, DefaultIPGateway, GatewayCostMetric,
DHCPEnabled, DHCPLeaseObtained, DHCPLeaseExpires, DHCPServer,
DNSServerSearchOrder
…
```

Příkazy jsou postupně vykonávány v tomto pořadí:

1. Získají se objekty vyhovující WQL dotazu uvedenému za přepínačem -query. To znamená, že se získají adaptéry typu Ethernet 802.3. Je však podivné, že do této kategorie spadají i adaptéry Bluetooth a Wi-Fi. Tuto specifikaci typu adaptéru lze bez problémů vynechat, slouží pouze jako filtr pro většinu virtuálních adaptérů.

2. Objekty síťových adaptérů vstupují po jednom do cyklu, kde je vždy poslán ven potrubím a kde je získán příslušný objekt (identifikovaný stejnou hodnotou atributu Index) typu Win32_NetworkAdapterConfiguration, který obsahuje další informace, které potřebujeme získat. Tento objekt je také poslán na výstup do potrubí.

3. Oba objekty vstoupí do formátovacího cmdletu Format-List, díky kterému jsou na výstup vypsány jejich atributy.

Celý skript je v příloze (Příloha F).

#### 3.4.2.2 Skript readSystemInfo.ps1

Skript readSystemInfo.ps1 slouží k získání informací o počítači. Tyto informace jsou pak zobrazovány v programu PSConfig na záložce Systém. Využita je třída Win32_ComputerSystem. Celý skript je připojen v příloze (Příloha G). Hlavní část skriptu je tato kolona[2]:

```
Get-WmiObject -Class Win32_ComputerSystem |
Format-List PSComputerName, Status, BootupState,
PowerSupplyState, ThermalState, DaylightInEffect,
DNSHostName, Domain, DomainRole, EnableDaylightSavingsTime,
Manufacturer, Model, NumberOfLogicalProcessors, NumberOfProcessors,
PCSystemType, SystemType, TotalPhysicalMemory
```

Tento skript je složen pouze ze dvou příkazů, které se vykonají takto:

1. Příkazem Get-WmiObject je získán objekt Win32_ComputerSystem, který je poslán do roury.
2. V cmdletu Format-List jsou vypsány specifikované atributy objektu Win32_ComputerSystem.

#### 3.4.2.3 Skript runScriptElevated.ps1

Tento skript slouží ke spuštění jiného skriptu s administrátorskými právy. Hlavní část skriptu je příkaz Start-Process. Ten má za úkol spustit další proces PowerShellu.

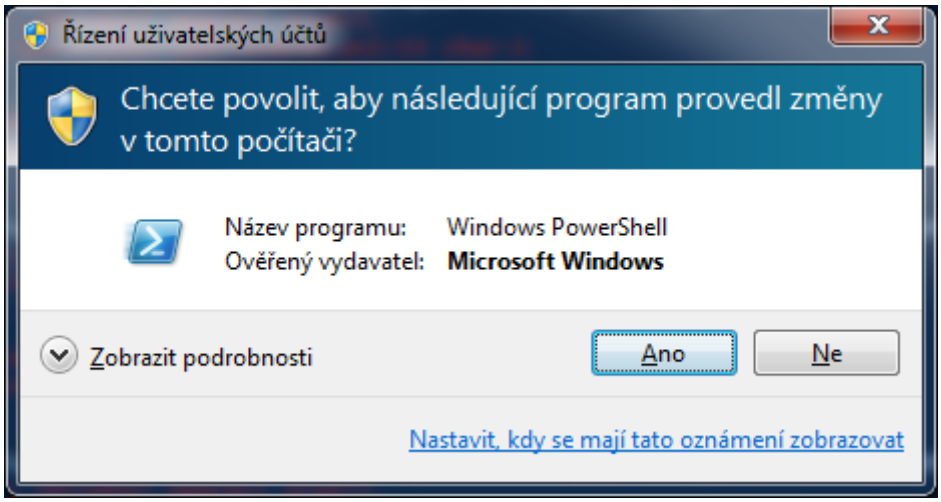
```
…
Start-Process "$PSHOME\powershell.exe" `
 -Verb Runas -ArgumentList "-command $script $arguments"
```

V tomto příkazu je za přepínačem -Verb uveden argument "Runas", což znamená, že proces bude spuštěn s vyššími právy [40]. Přepínač -ArgumentList specifikuje, s jakými argumenty bude proces PowerShellu spuštěn. Při provedení tohoto příkazu dojde k zavolání procesu PowerShellu, jako bychom ho volali z příkazové řádky tímto způsobem:

```
PS> $PSHOME\powershell.exe –Command skript argumenty_skriptu
```

[2] Kolona – Sled příkazů propojených přes roury [42]

Při provádění tohoto skriptu je vždy zobrazeno okno s požadavkem o potvrzení spuštění procesu PowerShellu s právy administrátora. To je vidět na obrázku (Obrázek 16):




**Obrázek 16 – Dotaz pro spuštění PowerShellu s právy administrátora**

Celý skript je obsažen v příloze (Příloha H).

#### 3.4.2.4 Skript setNetAdapter.ps1

Hlavní část skriptu setNetAdapter.ps1, sloužícímu k nastavení síťových adaptérů, je tato:

```
$changedAdapter =
Get-WmiObject -query `
$('select * from Win32_NetworkAdapterConfiguration where Index="{0}"' `
-f $index)
…
for($i = 0; $i -lt $methods.Count; $i++) {
      $method = $methods[$i]
      $parameter = $parameters[$i]

      if($parameter -ne $null) {
            $changedAdapter | %{$_.$method.Invoke($parameter)}
      } else {
            $changedAdapter | %{$_.$method.Invoke()}
      }
}
```

Tento kus kódu funguje tak, že je do proměnné $changedAdapter uložen objekt třídy Win32_NetworkAdapterConfiguration, sloužící ke konfiguraci síťových adaptérů. Poté jsou v cyklu načítány řetězce z pole $methods. Tyto řetězce reprezentují metody, které budou volány na proměnné $changedAdapter. Pole $parameters reprezentuje tzv. „zubaté

pole [3]“, které obsahuje příslušné parametry pro metody z pole $methods. Pro každou načtenou metodu $method je načteno i pole parametrů $parameter. V případě, že proměnná $parameter není rovna hodnotě $null se zavolá metoda $method s parametry $parameter. V opačném případě se zavolá metoda $method bez parametrů. Tímto způsobem lze zavolat jedním spuštěním skriptu i více metod s parametry. Při volání skriptu z příkazové řádky by příkaz vypadal například takto:

```
PS> .\setNetAdapter.ps1 -index 11 -methods EnableStatic, SetGateways
-parameters @(@('192.168.0.4', '255.255.255.0'), '192.168.0.1')
```

Ke změně nastavení síťových adaptérů je však potřeba oprávnění administrátora. Proto je potřeba, aby byl tento skript spouštěn přes skript runScriptElevated.ps1:

```
PS> .\runScriptElevated.ps1 -script ".\setNetAdapter.ps1" "-index 11
-methods EnableStatic, SetGateways -parameters @(@('192.168.0.4',
'255.255.255.0'), '192.168.0.1')"
```

Celý kód skriptu je obsažen v příloze (Příloha E).

[3] Zubaté pole – Pole, které obsahuje další pole s různými délkami. [43]

# Závěr

Hlavními dvěma cíly bakalářské práce bylo představit PowerShell z hlediska jeho možností a využití při správě počítače a porovnání s Bashem a vytvoření programu, který bude PowerShell využívat pro konfiguraci počítače.

Jazyk PowerShell je představen v první kapitole bakalářské práce. Jsou probrány základy použití jazyka včetně ukázek a k absolutní většině z nich je uveden i ekvivalent v jazyce Bash. Tento text lze bez problémů využít k naučení se základů jazyka PowerShell. V češtině není mnoho materiálů pojednávajícím o PowerShellu a už vůbec žádný, který by se zabýval i srovnáním s Bashem. Případní zájemci o PowerShell se tak mohou dozvědět i informace týkající se jiného operačního systému.

Aplikace je plně funkční a lze ji využít pro pohodlnější správu systému. Zobrazuje informace, ke kterým je složité se dostat, jako jsou například hodnoty stavu napájení počítače a role v doméně. Navíc i na jednom místě zobrazuje informace, které často nelze přehledně zobrazit a pozměňovat. Například přehled nastavení síťových adaptérů lze zobrazit na jednom místě pouze příkazem ipconfig v konzoli, což za prvé není moc přátelské pro uživatele zvyklé pouze na grafické rozraní, za druhé neumožňuje měnit nastavení síťových adaptérů. Pokud se tedy uživatel snaží nastavit síťový adaptér přes grafické menu, lze zase pro změnu upravovat hodnoty pouze u jednoho z nich a nemůže se tak zároveň podívat na nastavení jiných. Také nemůže pohodlně používat hodnot předem nastavených přes DHCP. V aplikaci PSConfig je možné si nechat potřebné nastavení, jako je například maska podsítě nebo výchozí brána, přiřadit přes DHCP a pak změnit pouze nějakou maličkost, jako je například adresa síťového adaptéru.

PowerShell je velice povedeným shellem, který se může směle porovnávat i s jinými, jako je například Bash, a v mnoha ohledech je i předčí. Oproti Bashi vyniká daleko přehlednější syntaxí a je mnohem přívětivější a jednodušší na použití. Avšak vzhledem k povaze systému Windows a objektově orientovanému přístupu je potřeba některé operace provádět mnohem složitějším způsobem, než je tomu například u textově orientovaného Bashe. To je ale spíše záležitostí operačního systému. Na systémech Windows jde o nenahraditelného pomocníka, jehož ovládnutí může usnadnit mnoho práce.

## Literatura


[1] techterms.com, „Shell,“ techterms.com, © 2013. [Online]. Available: http://www.techterms.com/definition/shell. [Přístup získán 24 Duben 2013].

[2] Free Software Foundation, „Bash Reference Manual: What is Bash?,“ Free Software Foundation, 22 August 2012. [Online]. Available: http://www.gnu.org/software/bash/manual/bashref.html#What-is-Bash_003f. [Přístup získán 24 Duben 2013].

[3] webopedia.com, „interpreter,“ QuinStreet Inc., © 2013. [Online]. Available: http://www.webopedia.com/TERM/I/interpreter.html. [Přístup získán 24 Duben 2013].

[4] webopedia.com, „CLI,“ QuinStreet Inc., © 2013. [Online]. Available: http://www.webopedia.com/TERM/C/CLI.html. [Přístup získán 24 Duben 2013].

[5] Microsoft, „Windows PowerShell Cmdlets,“ Microsoft, 9 Srpen 2012. [Online]. Available: http://technet.microsoft.com/en-us/library/bb648597%28v=vs.85%29.aspx. [Přístup získán 24 Duben 2013].

[6] Microsoft, „Overview of the .NET Framework,“ Microsoft, © 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. [Přístup získán 24 Duben 2013].

[7] Distributed Management Task Force, Inc., „Web-Based Enterprise Management,“ Distributed Management Task Force, Inc., 2013. [Online]. Available: http://dmtf.org/standards/wbem. [Přístup získán 24 Duben 2013].

[8] Microsoft, „About WMI,“ Microsoft, 19 Listopad 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa384642%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[9] T. WELTNER, Mastering PowerShell, BBS Technologies, 2009.

[10] T. LEE, K. MITSCHKE, M. E. SCHILL a T. TANASOVSKI, Windows PowerShell 2.0 Bible, 1st ed. editor, Indianapolis, IN: Wiley Pubishing, Inc., 2011.

[11] P. MALINA, Jak vyzrát na Windows PowerShell 2.0, 1. vydání editor, Brno: Computer Press, 2010.

[12] E. WILSON, Microsoft Windows PowerShell step by step, Redmond, WA: Microsoft Press, 2007.

[13] R. BLUM, Linux command line and shell scripting bible, Indianapolis, IN: Wiley Publishing, Inc., 2008.

[14] L. H. TOWER, „gnu.announce," 6 Srpen 1989. [Online]. Available: http://groups.google.com/group/gnu.announce/msg/a509f48ffb298c35?hl=en. [Přístup získán 24 Duben 2013].

[15] C. RAMEY, „The Bourne-Again Shell," [Online]. Available: http://www.aosabook.org/en/bash.html. [Přístup získán 24 Duben 2013].

[16] ormaaj, „Obsolete and deprecated syntax," bash-hackers.org, 23 Březen 2013. [Online]. Available: http://wiki.bash-hackers.org/scripting/obsolete. [Přístup získán 2 Květen 2013].

[17] Microsoft, „about_Redirection," Microsoft, 9 Srpen 2012. [Online]. Available: http://technet.microsoft.com/en-us/library/hh847746.aspx. [Přístup získán 2 Květen 2013].

[18] D. M. RITCHIE, „The Evolution of the Unix Time-sharing System: Pipes," Lucent Technologies Inc., © 1996. [Online]. Available: http://cm.bell-labs.com/cm/cs/who/dmr/hist.html#pipes. [Přístup získán 2 Květen 2013].

[19] tuxradar.com, „Exploring filters and pipes," tuxradar.com, 31 Březen 2009. [Online]. Available: http://www.tuxradar.com/content/exploring-filters-and-pipes. [Přístup získán 2 Květen 2013].

[20] Techotopia.com, „Windows PowerShell 1.0 Pipes and Redirection," Techotopia.com, 15 Květen 2009. [Online]. Available: http://www.techotopia.com/index.php/Windows_PowerShell_1.0_Pipes_and_Redirection. [Přístup získán 2 Květen 2013].

[21] J. D. COOK, „Comparing the Unix and PowerShell pipelines," The Endeavour, 9 Červen 2009. [Online]. Available: http://www.johndcook.com/blog/2009/06/09/comparing-the-unix-and-powershell-pipelines/. [Přístup získán 2 Květen 2013].

[22] Free Software Foundation, „Bash Reference Manual: Arrays," Free Software Foundation, 22 Srpen 2012. [Online]. Available: http://www.gnu.org/software/bash/manual/bashref.html#Arrays. [Přístup získán 2 Květen 2013].

[23] M. COOPER, „Advanced Bash-Scripting Guide: Bash, version 2," Listopad 2012. [Online]. Available: http://tldp.org/LDP/abs/html/bashver2.html. [Přístup získán 25 Duben 2013].

[24] Free Software Foundation, „Bash ftp,“ Free Software Foundation, [Online]. Available: http://ftp.gnu.org/gnu/bash/. [Přístup získán 25 Duben 2013].

[25] M. COOPER, „Advanced Bash-Scripting Guide: Bash, version 4,“ Listopad 2012. [Online]. Available: http://tldp.org/LDP/abs/html/bashver2.html. [Přístup získán 25 Duben 2013].

[26] PowerShellPro.com, „PowerShell Tutorial 10: Functions and Filters,“ PowerShellPro.com, © 2007-2011. [Online]. Available: http://www.powershellpro.com/powershell-tutorial-introduction/powershell-functions-filters/. [Přístup získán 25 Duben 2013].

[27] Microsoft, „Three Things You Might Not Know About Windows PowerShell Functions,“ Microsoft, © 2013. [Online]. Available: http://technet.microsoft.com/en-us/library/ff730957.aspx. [Přístup získán 25 Duben 2013].

[28] D. JONES, „Windows PowerShell: Defining Parameters,“ Microsoft, Srpen 2012. [Online]. Available: http://technet.microsoft.com/en-us/magazine/jj554301.aspx. [Přístup získán 25 Duben 2013].

[29] I. SHIELDS, „Linux tip: Bash test and comparison functions,“ IBM, 20 Únor 2007. [Online]. Available: http://www.ibm.com/developerworks/library/l-bash-test/. [Přístup získán 25 Duben 2013].

[30] M. COOPER, „Advanced Bash-Scripting Guide: Appendix B. Reference Cards,“ Listopad 2012. [Online]. Available: http://tldp.org/LDP/abs/html/refcards.html#AEN22006. [Přístup získán 25 Duben 2013].

[31] Microsoft, „File Class,“ Microsoft, © 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.io.file%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[32] Microsoft, „StreamReader Class,“ Microsoft, © 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.io.streamreader%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[33] Microsoft, „about_Break,“ Microsoft, 9 Srpen 2012. [Online]. Available: http://technet.microsoft.com/en-us/library/hh847873.aspx. [Přístup získán 25 Duben 2013].

[34] Microsoft, „Running Windows PowerShell Scripts,“ Microsoft, © 2013. [Online]. Available: http://technet.microsoft.com/en-us/library/ee176949.aspx. [Přístup získán 2 Květen 2013].

[35] PowerShell Team, „How does the RemoteSigned execution policy work?,“ Microsoft, 6 Březen 2007. [Online]. Available: http://blogs.msdn.com/b/powershell/archive/2007/03/07/how-does-the-remotesigned-execution-policy-work.aspx. [Přístup získán 2 Květen 2013].

[36] Microsoft, „Windows Management Instrumentation,“ Microsoft, 19 Listopad 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[37] Microsoft, „Common Information Model,“ Microsoft, 19 Listopad 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa389234%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[38] Distributed Management Task Force, Inc., „CIM Tutorial: Specifying Schema,“ Distributed Management Task Force, Inc., ©1999. [Online]. Available: http://www2.informatik.hu-berlin.de/~xing/Lib/cim-tutorial/extend/spec.html. [Přístup získán 25 Duben 2013].

[39] Microsoft, „WQL,“ Microsoft, 19 Listopad 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa394606%28VS.85%29.aspx. [Přístup získán 25 Duben 2013].

[40] A. DRISCOLL, „Adam Driscoll’s Favorite PowerShell Tips & Tricks,“ PowerShell Magazine, 11 Červen 2012. [Online]. Available: http://www.powershellmagazine.com/2012/06/11/adam-driscolls-favorite-powershell-tips-tricks/. [Přístup získán 2 Květen 2013].

[41] Microsoft, „Win32_NetworkAdapter class,“ Microsoft, 19 Listopad 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa394216%28v=vs.85%29.aspx. [Přístup získán 25 Duben 2013].

[42] M. BRANDEJS, „PV004 – UNIX,“ 3 Únor 2009. [Online]. Available: http://www.fi.muni.cz/usr/brandejs/unix/brandejs_unix_print.pdf. [Přístup získán 2 Květen 2013].

[43] Microsoft, „Jagged Arrays,“ Microsoft, © 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/2s05feca.aspx. [Přístup získán 2 Květen 2013].

## Příloha A – Nejpoužívanější akce cmdletů

Tato tabulka představuje přehled nejpoužívanějších akcí cmdletů.

| Akce | Popis |
|---|---|
| **Add** | Přidání |
| **Clear** | Smazání |
| **Compare** | Porovnání |
| **Convert** | Konverze |
| **Copy** | Kopírování |
| **Export** | Exportování |
| **Format** | Formátování |
| **Get** | Získání |
| **Group** | Seskupování |
| **Import** | Importování |
| **Measure** | Změření |
| **Move** | Přesunutí |
| **New** | Vytvoření nového |
| **Out** | Výstup |
| **Read** | Čtení |
| **Remove** | Odstranění |
| **Rename** | Přejmenování |
| **Resolve** | Vyřešení výrazu s wildcard znaky |
| **Restart** | Restartování |
| **Resume** | Obnovení |
| **Select** | Výběr |
| **Set** | Nastavení |
| **Sort** | Třídění |
| **Split** | Rozdělení |
| **Start** | Spuštění |
| **Stop** | Zastavení |
| **Suspend** | Pozastavení |
| **Tee** | Rozdělení výstupu |
| **Test** | Testování |
| **Trace** | Trasování |
| **Update** | Aktualizování |
| **Write** | Zapsání |

Zdroj: [9]

## Příloha B – Tabulka aliasů

Tato tabulka obsahuje ukázku aliasů, které jsou nastavené v PowerShellu již v základu. Z těchto aliasů je vidět, že v PowerShellu lze používat příkazy známé i z prostředí UNIX, respektive GNU/Linux.

| Alias | Skutečný cmdlet |
|---|---|
| **cat** | Get-Content |
| **cd** | Set-Location |
| **clear** | Clear-Host |
| **cp** | Copy-Item |
| **echo** | Write-Output |
| **kill** | Stop-Process |
| **ls** | Get-ChildItem |
| **man** | help |
| **mv** | Move-Item |
| **pwd** | Get-Location |
| **rm** | Remove-Item |
| **sort** | Sort-Object |

## Příloha C – Operátory porovnání

Pro označení operátorů porovnávajících číselné hodnoty v hranatých závorkách je použito písmeno n, pro operátory porovnání číselných hodnot ve dvojitých kulatých závorkách je použito písmeno d a pro označení operátorů porovnávajících řetězce je použito písmeno s.

| PowerShell | Bash | Běžný operátor | Popis |
|---|---|---|---|
| **-eq** | **n: -eq, d: ==, s: =** | **=, ==** | Rovnost |
| **-ne** | **n: -ne, d: !=, s: !=** | **!=, <>** | Nerovnost |
| **-gt** | **n: -gt, d: >, s: >** | **>** | Větší než |
| **-ge** | **n: -ge, d: >=** | **>=** | Větší nebo rovno než |
| **-lt** | **n: -lt, d: <, s: <** | **<** | Menší než |
| **-le** | **n: -le, d: <=** | **<=** | Menší nebo rovno než |
| **-Like** | | | Porovnání řetězce vůči řetězci s wildcard znaky |
| **-NotLike** | | | Stejné jako Like, ale výsledek znegován |
| **-Match** | | | Vyhledání v řetězci podle regulárního výrazu |
| **-NotMatch** | | | Jako Match, ale výsledek znegovaný |
| **-Contains** | | | Obsahuje (v kolekci) |
| **-NotContains** | | | Neobsahuje (v kolekci) |
| **-In** | | | Test jestli je prvek v kolekci |
| **-NotIn** | | | Jako In, ale výsledek znegován |
| **-Replace** | | | Nahrazuje specifikovaný prvek jiným. |

## Příloha D – Skript foreach_foreachobject.ps1

Tento skript slouží ke změření rychlosti cyklů Foreach a ForEach-Object v PowerShellu. Funguje tak, že je definováno pole o tisíci prvcích a cykly pak provádí čtení z tohoto pole. Operace přečtení celého pole je provedena pro oba příkazy padesátkrát a z výsledných hodnot je vytvořena hodnota průměrná. Statistiky jsou poté vypsány na obrazovku.

```
# skript foreach_foreachobject.ps1
# Proměnná $a je pole čísel od 1 do 1000, které se bude v cyklu vypisovat
# Každý výpis pole od 1 do 1000 se provede 50 a výsledný čas se zaznamená
# Z časů 50-ti provedených iterací se vypočítá průměr

$a = 1..1000

$ForEachObject = @()
For ($i = 0; $i -lt 50; $i++) {
      $ForEachObject+=(
            Measure-Command {
                  $a | ForEach-Object { $_ }
            }
      ).TotalMilliseconds
}
$vysledky = $ForEachObject | Measure-Object -Average
Write-Host "Počet opakování: $($vysledky.Count) a průměrný čas pro For-
EachObject: $($vysledky.Average)"


$ForeachCommand = @()
For ($i = 0; $i -lt 50; $i++) {
      $ForEachCommand+=(
            Measure-Command {
                  Foreach ($x in $a) { $x }
            }
      ).TotalMilliseconds
}
$vysledky = $ForeachCommand | Measure-Object -Average
Write-Host "Počet opakování: $($vysledky.Count) a průměrný čas pro
Foreach: $($vysledky.Average)"
```

## Příloha E – Skript setNetAdapter.ps1

Tento skript slouží k nastavení síťových adaptérů. Předávají se mu tři argumenty. Prvním je index síťového adaptéru, druhým je pole řetězců s názvy metod, které se mají dynamicky zavolat na třídě konfigurující příslušný síťový adaptér, a třetím je vícerozměrné pole, které obsahuje argumenty ke zmíněným metodám. Skript kontroluje, aby byl počet metod v proměnné $methods stejný jako počet polí obsažených v proměnné $parameters.

```
# Skript setNetAdapter.ps1

# Definice parametrů skriptu.
Param
(
	$index=$(throw "Chybějící parametr -index"),
	[string[]]$methods,
	$parameters
)

# Získání objektu pro konfiguraci síťového adaptéru.
# Pro získání objektu je použit jazyk WQL.
# Do WQL řetězce je namísto symbolu {0} dosazena hodnota $index.
$changedAdapter =
Get-WmiObject -query `
$('select * from Win32_NetworkAdapterConfiguration where Index="{0}"' `
-f $index)

# Proměnná uchovávající počet parametrů.
$numberOfParams = 0;

# Cyklus, který počítá počet prvků v parametru $parameters.
# Počet parametrů se zvýší o hodnotu 1, pokud prvek existuje a nerovná se
# řetězci "null".
for($i=0; $i -lt $parameters.Count; $i++) {
	if($parameters[$i] -ne "null") {
		$numberOfParams++
	}
}

# Ověření, že počet metod a počet parametrů těchto metod je stejný.
# Pokud tento počet není stejný, je vyvolána výjimka.
if($methods.Count -ne $numberOfParams) {
	throw "Chyba! Počet funkcí a jejich parametrů je odlišný!"
}

# Cyklus, který všechny řetězce "noParam" v poli $parameters nahradí
# za nulovou hodnotu.
for($i = 0; $i -lt $parameters.Count; $i++) {
	if($parameters[$i] -eq "noParam") {
		$parameters[$i] = $null
	}
}



# Cyklus, který prochází všechny řetězce v poli $methods.
```

```
# Pro každý prvek tohoto pole zavolá metodu s názvem odpovídajícím
# danému řetězci v poli.
for($i = 0; $i -lt $methods.Count; $i++) {
	$method = $methods[$i]
	$parameter = $parameters[$i]

	# Pokud má parametr hodnotu $null, zavolá se metoda bez parametrů,
	# jinak se předají parametry v proměnné $parametr.
	if($parameter -ne $null) {
		$changedAdapter | %{$_.$method.Invoke($parameter)}
	} else {
		$changedAdapter | %{$_.$method.Invoke()}
	}
}
```

## Příloha F – Skript readNetworkAdaptersAllInfo.ps1

Tento skript načítá informace o síťových adaptérech.

```
# Skript readNetworkAdaptersAllInfo.ps1

# Uložení současné velikosti bufferu PowerShellu.
$oldBufferSize=$host.UI.RawUI.BufferSize

# Nastavení velikosti bufferu na větší velikost.
# Zabrání se tak zalamování dlouhých řádků.
$host.UI.RawUI.BufferSize =
new-object System.Management.Automation.Host.Size(512,50)

# Získání všech objektů Win32_NetworkAdapter.
Get-WmiObject -query `
'select * from Win32_NetworkAdapter where AdapterType="Ethernet 802.3"' |
ForEach-Object {
      # Výstup současného síťového adaptéru
      # na standardní výstup příkazu ForEach-Object.
      $_

      # Získání objektu Win32_NetworkAdapterConfiguration,
      # který odpovídá současnému síťovému adaptéru
      # a předání na standardní výstup příkazu ForEach-Object.
      $(Get-WmiObject -query `
      $('select * from Win32_NetworkAdapterConfiguration where
Index="{0}"' -f $_.Index))

      # Zobrazení vybraných informací u předchozích dvou objektů.
} | Format-List Index, Name, NetEnabled, NetConnectionID, MACAddress,
Manufacturer, IPAddress, IPSubnet, DefaultIPGateway, GatewayCostMetric,
DHCPEnabled, DHCPLeaseObtained, DHCPLeaseExpires, DHCPServer,
DNSServerSearchOrder

# Obnovení původní hodnoty velikosti bufferu.
$host.UI.RawUI.BufferSize = $oldBufferSize
```

## Příloha G – Skript readSystemInfo.ps1

Skript načítající informace o systému.

```
# Skript readSystemInfo.ps1

# Získání objektu Win32_ComputerSystem.
Get-WmiObject -Class Win32_ComputerSystem |

# Zobrazení vybraných informací o objektu Win32_ComputerSystem.
Format-List PSComputerName, Status, BootupState,
PowerSupplyState, ThermalState, DaylightInEffect,
DNSHostName, Domain, DomainRole, EnableDaylightSavingsTime,
Manufacturer, Model, NumberOfLogicalProcessors, NumberOfProcessors,
PCSystemType, SystemType, TotalPhysicalMemory
```

## Příloha H – Skript runScriptElevated.ps1

Skript, který spustí jiný skript s právy administrátora.

```
# Skript runScriptElevated.ps1

# Deklarace paramterů skriptu.
# Parametr $script je povinný - pokud chybí,
# je vyvolána výjimka.
Param
(
      [string]$script=$(throw "Chybějící parametr -script"),
      $arguments
)

# Spuštění procesu PowerShell s právy administrátora.
# Jako parametr je mu předán skript s parametry.
Start-Process "$PSHOME\powershell.exe" `
 -Verb Runas -ArgumentList "-command $script $arguments"
```